```
/*
    Andrew Maclean
    March 15th, 2017
    CS 4110 Compiler

    program     -> blockst
    blockst     -> BEGINTOK stats ENDTOK
    stats       -> statmt ';' stats | empty
    decl        -> BASICTYPETOK IDTOK
    statmt      -> decl | ifstat | assstat |  blockst | loopst | iostat | empty
    assstat     -> idref ASTOK expression
    ifstat      -> IFTOK  expression THENTOK statmt
    loopst      -> WHILETOK expression DOTOK statmt
    iostat      -> READTOK ( idref ) | WRITETOK (expression)
    expression  -> term expprime
    expprime    -> ADDOPTOK  term expprime | empty
    term        -> relfactor termprime
    termprime   -> MULOPTOK  relfactor termprime | empty
    relfactor   -> factor factorprime
    factorprime -> RELOPTOK factor | empty
    factor      -> NOTTOK factor | idref | LITTOK | '(' expression ')'
    idref       -> IDTOK


    READ / WRITE doens't appear to work correctly
    I'm not sure what write line does, exactly
    Assembly code needs optimization

*/


#ifndef PARSER_H
#define PARSER_H

#include <iostream>
#include <fstream>
#include <string>
#include <queue>
#include "SymbolTable.h"

using namespace std;

struct token
{
    string tokenName;
    int lineNo;
    int tokenNumber;
};

class Parser
{
public:
    Parser(queue<token> tokenList)
    {
        tokens = tokenList;
        identifiers = new HashTable();
        assem.open("assemblyCode.txt");
        err.open("errors.txt");
        currentAddr = 0;
        program();
        assem.close();
        err.close();
        currTemp = 0;
    }

    // makes sure there is at least one token left to pop
    void countCheck()
    {
        if (tokens.size() == 0)
        {
            err << "\nNot enough tokens" << endl;
            system("pause");
            exit(EXIT_FAILURE);
        }
    }

    //does prints the rule number somewhere
    void print(int tok)
    {
        //cout << tok << " ";
    }
    //idref -> IDTOK
    void idref(bool left)
    {
        print(31);

        //as long as the symbol table finds the identifier...
        if (identifiers->find(tokens.front().tokenName)->name != "Not Found")
        {
            //doesn't write any assembly for lefthand identifiers
            if (!left)
            {
                //pulls an identifiers and puts it in the next spot in the stack
                assem << "lw $t" << currTemp << " " << identifiers->find
                  (tokens.front().tokenName)->address << " # load value\n";
                assem << "sw $t" << currTemp << " " << currentAddr << endl;
                currentAddr -= 4;
            }

            tokens.pop(); //pop IDTOK
        }

        //give an error otherwise
        else
        {
            err << "\nIdentifier " << tokens.front().tokenName << " was not found on line: " << tokens.front().lineNo << endl;
            tokens.pop();
        }

        countCheck();
    }

    //factor -> NOTTOK factor | idref | LITTOK | '(' expression ')'
    void factor()
    {
        // token 17 is NOTTOK
        if (tokens.front().tokenNumber == 17)
        {
            print(27);
            tokens.pop(); //pop NOTTOK
            factor();
        }
        //token 1 is ID
        else if (tokens.front().tokenNumber == 1)
        {
            print(28);
            idref(false);
        }
        //token 2 is LITERAL
        else if (tokens.front().tokenNumber == 2)
        {
            currTemp++;
            print(29);
            //don't follow this with strings
            if (tokens.front().tokenName[0] != '"')
            {
                // if the token is a number
                if (isdigit(tokens.front().tokenName[0]))
                    assem << "li $t" << currTemp << ", " << tokens.front().tokenName << " # assinging an int value" << endl;

                // if it's false
                else if (tokens.front().tokenName[0] == 'F')
                    assem << "li $t" << currTemp << ", 0 # assigning a false value\n";

                // if it's true
                else if (tokens.front().tokenName[0] == 'T')
                    assem << "li $t" << currTemp << ", 1 # assigning a true value\n";

                // store in next available spot in stack
                assem << "sw $t" << currTemp << ", " << currentAddr << endl;
            }

            tokens.pop(); //pop LITTOK
            countCheck();
            currentAddr -= 4;
        }
        //token 14 is (
        else if (tokens.front().tokenNumber == 14)
        {
            print(30);

            tokens.pop(); //pop (

            countCheck();
            expression();

            //token 15 is )
            if (tokens.front().tokenNumber != 15) // missing token error
                err << "\nExpected \')\' on line: " << tokens.front().lineNo << 
                  endl;
            else
                tokens.pop(); //pop )
        }
        countCheck();

    }

    //factorprime -> RELOPTOK factor | empty
    void factorprime()
    {
        string temp;
        //token 6 is RELOPTOK
        if (tokens.front().tokenNumber == 6)
        {
            print(25);
            // saves the token for later
            temp = tokens.front().tokenName;
            tokens.pop(); //pop RELOPTOK
            countCheck();
            factor();

            currentAddr += 4;
            currTemp = 0;

            // compares and goes to skip
            assem << "lw $t" << currTemp << " " << currentAddr << endl;
            currentAddr += 4;
            currTemp++;
            assem << "lw $t" << currTemp << " " << currentAddr << endl;

            if (temp == "<")
                assem << "blt $t" << currTemp << " $t" << currTemp - 1 << " Skip" 
                   << endl;

            else if (temp == ">")
                assem << "bgt $t" << currTemp << " $t" << currTemp - 1 << " Skip" 
                   << endl;

            else if (temp == "!=")
                assem << "bne $t" << currTemp << " $t" << currTemp - 1 << " Skip" 
                   << endl;

            assem << "sw $t" << currTemp << " " << currentAddr << endl;

            currTemp = 0;
        }
        else
            print(26);
    }

    //relfactor -> factor factorprime
    void relfactor()
    {
        print(24);
        factor();

        factorprime();
    }

    //termprime -> MULOPTOK relfactor termprime | empty
    void termprime()
    {
        string temp;
        //token 5 is MULOPTOK
        if (tokens.front().tokenNumber == 5)
        {
            print(22);
            temp = tokens.front().tokenName;
            tokens.pop(); //pop MULOPTOK
            countCheck();
            relfactor();
            currentAddr += 4;
            currTemp = 0;

            // loads two registers with the numbers to be operated
            assem << "lw $t" << currTemp << " " << currentAddr << endl;
            currentAddr += 4;
            currTemp++;
            assem << "lw $t" << currTemp << " " << currentAddr << endl;
            currTemp++;

            // computes the operation
            if (temp == "*")
                assem << "mult $t" << currTemp << " $t" << currTemp - 1 << " $t"  
                  << currTemp - 2 << endl;

            else if (temp == "AND")
                assem << "and $t" << currTemp << " $t" << currTemp - 1 << " $t" 
                  << currTemp - 2 << endl;

            else if (temp == "/" || temp == "DIV")
                assem << "div $t" << currTemp - 1 << " $t" << currTemp - 2 << 
                  endl;
                assem << "mflo $t" << currTemp << endl;

            // stores the result
            assem << "sw $t" << currTemp << " " << currentAddr << endl;

            currentAddr -= 4;
            currTemp = 0;
            termprime();

        }
        else
            print(23);
    }

    //term -> relfactor termprime
    void term()
    {
        print(21);
        relfactor();
        termprime();
    }

    //expprime -> ADDOPTOK term expprime | empty
    void expprime()
    {
        string temp;
        //token 4 is ADDOPTOK
        if (tokens.front().tokenNumber == 4)
        {
            print(19);
            temp = tokens.front().tokenName;
            tokens.pop(); //pop ADDOPTOK
            countCheck();
            term();

            //div1 and div2 are the two temps being worked on
            currTemp = 0;
            currentAddr += 4;

            assem << "lw $t" << currTemp << " " << currentAddr << endl;
            currTemp++;
            currentAddr += 4;
            assem << "lw $t" << currTemp << " " << currentAddr << endl;
            currTemp++;

```

```
                // mostly the same as termprime
                if (temp == "+")
                    assem << "add $t" << currTemp << " $t" << currTemp - 1 << " $t"
                      << currTemp - 2 << endl;

                else if (temp == "-")
                    assem << "sub $t" << currTemp << " $t" << currTemp - 1 << " $t"
                      << currTemp - 2 << endl;

                else if (temp == "OR")
                    assem << "or $t" << currTemp << " $t" << currTemp - 1 << " $t" <<
                       currTemp - 2 << endl;

                assem << "sw $t" << currTemp << " " << currentAddr << endl;

                currentAddr -= 4;
                currTemp = 0;
                expprime();

            }
            else
                print(20);
        }

        //expression -> term expprime
        void expression()
        {
            print(18);
            term();
            expprime();
        }

        //iostat -> READTOK '(' idref ')' | WRITETOK '(' expression ')'
        void iostat()
        {
            string key;

            key = tokens.front().tokenName;
            tokens.pop(); //pop READTOK or WRITETOK
            countCheck();

            if (tokens.front().tokenNumber != 14) // missing token error, (
                err << "\nExpected \'(\' in iostat on line: " << tokens.front
                  ().lineNo << endl;
            else
                tokens.pop(); //pop open parenthesis

            countCheck();

            if (key == "READ")
            {
                // reads an input
                print(16);
                idref(false);

            }
            else
            {
                print(17);
                // writeline function
                if (key == "WRITELN")
                {
                    // writes a string
                    if (identifiers->find(tokens.front().tokenName)->type == "string")
                    {
                        assem << "output: .asciiz \"\\n" << tokens.front().tokenName << "\\n\"" << endl;
                        assem << "la $a0 output" << endl;
                        assem << "li $v0 4" << endl;
                    }
                    // writes a variable from memory
                    else
                    {
                        assem << "lw $a0 " << currentAddr << endl;
                        assem << "li $v0 1" << endl;
                    }
                    expression();
                    assem << "syscall" << endl;
                }
                // regular write function
                else
                {
                    if (identifiers->find(tokens.front().tokenName)->type == "string")
                    {
                        assem << "output: .asciiz \"" << tokens.front().tokenName << "\"" << endl;
                        assem << "la $a0 output" << endl;
                        assem << "li $v0 4" << endl;
                    }
                    else
                    {
                        assem << "lw $a0 " << currentAddr << endl;
                        assem << "li $v0 1" << endl;
                    }
                    expression();
                    assem << "syscall" << endl;
                }

            }

            if (tokens.front().tokenNumber != 15) // missing token error on )
                err << "\nExpected \')\' in iostat on line: " << tokens.front
```

```
                ().lineNo << endl;
        else
            tokens.pop(); //pop closed parentheis

        countCheck();
    }

    //loopst -> WHILETOK expression DOTOK statmt
    void loopst()
    {
        print(15);
        // WHILE is token 11
        tokens.pop(); //pop WHILETOK
        assem << "Loop: ";
        expression();
        //DO is token 12
        if (tokens.front().tokenNumber != 12) // missing token error
            err << "\nExpected \"DO\" token, got " << tokens.front().tokenName << " on line: " << tokens.front().lineNo << endl;
        else
            tokens.pop(); //pop DOTOK
        countCheck();

        statmt();
        assem << "j Loop " << endl;
        assem << "Skip: ";
    }

    //ifstat -> IFTOK expression THENTOK statmt
    void ifstat()
    {
        print(14);

        tokens.pop(); //pop IFTOK

        countCheck();
        expression();

        if (tokens.front().tokenNumber != 10) // missing token error, token 10 is THEN
            err << "\nExpected \"THEN\" token, got " << tokens.front().tokenName << " on line: " << tokens.front().lineNo << endl;
        else
            tokens.pop(); //pop THENTOK

        countCheck();
        statmt();
        // the position to skip to
        assem << "Skip: ";
    }

    //assstat -> idref ASTOK expression
    void assstat()
    {
        // loc is the offset of a variable
        int loc;
        //as long as the symbol table finds the identifier...
        if (identifiers->find(tokens.front().tokenName)->name != "Not Found")
        {
            Data *data = identifiers->find(tokens.front().tokenName);
            loc = data->address;
            bool left = true;
            idref(left);
        }
        // clears the code and declares an error
        else
        {
            err << "\nUninitialized variable " << tokens.front().tokenName << " on line: " << tokens.front().lineNo << endl;
            assem.close();
            assem.open("assemblyCode.txt");
            assem << "";
            assem.close();
            exit(EXIT_FAILURE);
        }
        if (tokens.front().tokenNumber != 19) // missing token error ASTOK
            err << "\nExpected :=, got " << tokens.front().tokenName << " on line: " << tokens.front().lineNo << endl;
        else
            tokens.pop(); // pops assignment token
        //countCheck makes sure that there is at least one token left
        countCheck();
        expression();
        currentAddr += 4;
        // stores the last spot in memory to a variable
        assem << "lw $t" << currTemp << " " << currentAddr << endl;
        assem << "sw $t" << currTemp << " " << loc << " # assign to variable" << endl;
        currTemp = 0;
    }

    //stmt -> decl | ifstat | assstat | blockst | loopst | iostat | empty
    void statmt()
    {
        if (tokens.front().tokenNumber == 3)
        {
            print(6);
            decl();
        }
        else if (tokens.front().tokenNumber == 9)
        {
            print(7);
            ifstat();
        }
        else if (tokens.front().tokenNumber == 1)
        {
            print(8);
            assstat();
        }
        else if (tokens.front().tokenNumber == 7)
        {
            print(9);
            blockst();
        }
        else if (tokens.front().tokenNumber == 11)
        {
            print(10);
            loopst();
        }
        else if (tokens.front().tokenNumber == 13)
        {
            print(11);
            iostat();
        }
        else
            print(12);
    }

    //decl -> BASICTYPETOK IDTOK
    void decl()
    {
        print(5);
        // the token type before it gets popped
        string type = tokens.front().tokenName;
        tokens.pop(); //pop TYPETOK

        countCheck();

        if (tokens.front().tokenNumber != 1) // missing token error IDENT
            err << "\nMissing identifier on line: " << tokens.front().lineNo << 
              endl;
        else
        {
            // fills the symbol table with the new identifier
            if (identifiers->find(tokens.front().tokenName)->name == "Not Found")
            {
                Data inserted = { type, tokens.front().tokenName, currentAddr};
                identifiers->insert(inserted);
                assem << "# stored a new variable in the next offset" << endl;
                tokens.pop(); // pop IDTOK
            }
            //doesn't make two identifiers
            else
            {
                err << "\nIdentifier already exists, on line: " << tokens.front
                  ().lineNo << endl;
                tokens.pop();
            }
        }
        countCheck();
        currentAddr -= 4;
    }

    //stats -> statmt ';' stats | empty
    void stats()
    {
        if (tokens.front().tokenNumber != 8)
        {
            print(3);
            statmt();
            if (tokens.front().tokenNumber != 16) // missing token error
                err << "\nMissing \';\' on line: " << tokens.front().lineNo << endl;
            else //pop ;
            {
                tokens.pop();
                countCheck();
                assem << "\n# new statement" << endl;
                stats();
            }
        }
        else
            print(4);
    }

    //blockst -> BEGINTOK stats ENDTOK
    void blockst()
    {
        print(2);
        if (tokens.front().tokenNumber != 7) // missing token error
            err << "\nMissing begin token on line: " << tokens.front().lineNo << endl;
        else
        {
            //pop BEGIN
            tokens.pop();
            identifiers->newScope();
        }
        countCheck(); // makes sure that the token count is greater than zero
        stats();
        if (tokens.front().tokenNumber != 8) // missing token error
            err << "\nMissing end token on line: " << tokens.front().lineNo << endl;
        else
        {
            //pop END
            tokens.pop();
            identifiers->closeScope();
        }
        countCheck();
    }

    //program -> blockst '.'
    void program()
    {
        assem << ".text\n.globl main\nmove $fp $sp\nla $a0 ProgStart\nli $v0 4\nsyscall\n\n";
        blockst();
        assem << "\nla $a0 ProgEnd\nli $v0 4\nsyscall\nli $v0 10\nsyscall\n.data\nProgStart: .asciiz \"Program Start\\n\"\nProgEnd: .asciiz \"Program End\\n\"";
        if (tokens.empty()) // missing token error
            err << "\nNo end of program token\n";
        cout << "\n";
        //prints the symbol table
        identifiers->print();
    }

private:
        queue<token> tokens;
        HashTable* identifiers;
        int currentAddr;
        ofstream assem;
        ofstream err;
        int currTemp;
};

#endif
```