lab5/L5_lab5plantilla.cpp

```cpp
#include <sstream>
#include <iostream>
#include <stdlib.h>
#include <cstring>

using namespace std;


class Token {
public:
    enum Type {PLUS, MINUS, MUL,DIV, NUM, ERR, PD, PI, END};
    Type type;
    string text;
    Token(Type);
    Token(Type, char c);
    Token(Type, const string& source, int first, int last);
};

class Scanner {
private:
    string input;
    int first, current;
public:
    Scanner(const char* in_s);
    Token* nextToken();
    ~Scanner();

};

enum BinaryOp { PLUS, MINUS,MUL,DIV };

class Exp {
public:
    virtual void print() = 0;
    virtual int eval() = 0;
    virtual ~Exp() = 0;
    static char binopToChar(BinaryOp op);
};


class BinaryExp : public Exp {
public:
    Exp *left, *right;
    BinaryOp op;
    BinaryExp(Exp* l, Exp* r, BinaryOp op);
    void print();
    int eval();
    ~BinaryExp();
};

class NumberExp : public Exp {
```

```cpp
public:
    int value;
    NumberExp(int v);
    void print();
    int eval();
    ~NumberExp();
};

class Parser {
private:
    Scanner* scanner;
    Token *current, *previous;
    bool match(Token::Type ttype);
    bool check(Token::Type ttype);
    bool advance();
    bool isAtEnd();
    Exp* parseExpression();
    Exp* parseTerm();
    Exp* parseFactor();
    bool tokenToOp(Token* tk, BinaryOp& op);
public:
    Parser(Scanner* scanner);
    Exp* parse();
};


Token::Token(Type type):type(type) { text = ""; }

Token::Token(Type type, char c):type(type) { text = c; }

Token::Token(Type type, const string& source, int first, int last):type(type) {
    text = source.substr(first,last);
}

std::ostream& operator << ( std::ostream& outs, const Token & tok )
{
    if (tok.text.empty())
        return outs << tok.type;
    else
        return outs << "TOK" << "(" << tok.text << ")";
}

std::ostream& operator << ( std::ostream& outs, const Token* tok ) {
    return outs << *tok;
}

// SCANNER //

Scanner::Scanner(const char* s):input(s),first(0), current(0) { }

Token* Scanner::nextToken() {
    Token* token;
    while (input[current]==' ') current++;
    if (input[current] == '\0') return new Token(Token::END);
    char c  = input[current];
    first = current;
    if (isdigit(c)) {
        current++;
        while (isdigit(input[current]))
            current++;
        token = new Token(Token::NUM,input,first,current-first);
    } else if (strchr("+-*/()", c)) {
        switch(c) {
            case '+': token = new Token(Token::PLUS,c); break;
            case '-': token = new Token(Token::MINUS,c); break;
            case '*': token = new Token(Token::MUL,c); break;
            case '/': token = new Token(Token::DIV,c); break;
            case '(': token = new Token(Token::PI,c); break;
            case ')': token = new Token(Token::PD,c); break;
            default: cout << "No deberia llegar aca" << endl;
        }
        current++;
    } else {
        token = new Token(Token::ERR, c);
        current++;
    }
    return token;
}

Scanner::~Scanner() { }

// PARSER //

bool Parser::match(Token::Type ttype) {
    if (check(ttype)) {
        advance();
        return true;
    }
    return false;
}

bool Parser::check(Token::Type ttype) {
    if (isAtEnd()) return false;
    return current->type == ttype;
}

bool Parser::advance() {
    if (!isAtEnd()) {
        Token* temp =current;
        if (previous) delete previous;
        current = scanner->nextToken();
        previous = temp;
        if (check(Token::ERR)) {
            cout << "Parse error, unrecognised character: " << current->text <<
endl;
            exit(0);
        }
        return true;
    }
    return false;
}

bool Parser::isAtEnd() {
    return (current->type == Token::END);
}

Parser::Parser(Scanner* sc):scanner(sc) {
    previous = current = NULL;
    return;
};

Exp* Parser::parse() {
    current = scanner->nextToken();
    if (check(Token::ERR)) {
        cout << "Error en scanner - caracter invalido" << endl;
        exit(0);
    }
    Exp* exp = parseExpression();
    if (current) delete current;
    return exp;
}

Exp* Parser::parseExpression() {

    Exp* left = parseTerm();

    while (match(Token::PLUS) || match(Token::MINUS)) {
        BinaryOp op;
        if (previous->type == Token::PLUS){
            op = PLUS;
        }
        else if (previous->type == Token::MINUS){
            op = MINUS;
        }
        Exp* right = parseTerm();
        left = new BinaryExp(left, right, op);
    }

    return left;
}

Exp* Parser::parseTerm() {

    Exp* left = parseFactor();

    while (match(Token::MUL) || match(Token::DIV)) {
        BinaryOp op;
        if (previous->type == Token::MUL){
            op = MUL;
        }
        else if (previous->type == Token::DIV){
            op = DIV;
        }
        Exp* right = parseFactor();
        left = new BinaryExp(left, right, op);
    }
    return left;
}

Exp* Parser::parseFactor() {
    Exp* e;
    if (match(Token::NUM)) {
        return new NumberExp(stoi(previous->text));
    }
    else if (match(Token::PI)){
        e = parseExpression();
        if (!match(Token::PD)){
            cout << "Falta parentesis derecho" << endl;
        }
        return e;
    }
    cout << "Error: se esperaba un número." << endl;
    exit(0);
}

char Exp::binopToChar(BinaryOp op) {
    char  c=' ';
    switch(op) {
        case PLUS: c = '+'; break;
        case MINUS: c = '-'; break;
        case MUL: c = '*'; break;
        case DIV: c = '/'; break;
        default: c = '$';
    }
    return c;
}

// AST //


BinaryExp::BinaryExp(Exp* l, Exp* r, BinaryOp op):left(l),right(r),op(op) {}
NumberExp::NumberExp(int v):value(v) {}

Exp::~Exp() {}
BinaryExp::~BinaryExp() { delete left; delete right; }
NumberExp::~NumberExp() { }


void BinaryExp::print() {
    left->print();
    char c = binopToChar(this->op);;
    cout << ' ' << c << ' ';
    right->print();
}



void NumberExp::print() {
    cout << value;
}

int BinaryExp::eval() {
    int result;
    int v1=left->eval();
    int v2=right->eval();
    switch(this->op) {
        case PLUS: result = v1+v2; break;
        case MINUS: result = v1-v2; break;
        case MUL: result = v1*v2; break;
        case DIV: result = v1/v2; break;
        default:
            cout << "Operador desconocido" << endl;
            result = 0;
    }
    return result;
}

int NumberExp::eval() {
    return value;
}


void test_scanner(Scanner * scanner) {
    Token* current;
    current = scanner->nextToken();
    while (current->type != Token::END) {
        if (current->type == Token::ERR) {
            cout << "Error en scanner - caracter invalido: " << current->text <<
endl;
            break;
        } else
            cout << current << endl;
        current = scanner->nextToken();
    }
    exit(1);

}

int main(int argc, const char* argv[]) {

    if (argc != 2) {
        cout << "Incorrect number of arguments" << endl;
        exit(1);
    }

    Scanner scanner(argv[1]);

    //test_scanner(&scanner);

    Parser parser(&scanner);

    Exp *exp = parser.parse();

    cout << "expr: ";
    exp->print();
    cout << endl;

    cout << "eval: ";
    cout << exp->eval() << endl;
    // linux
    // cd ".\Compiladores\"
    // g++ L5_lab5plantilla.cpp -o lab5eP
    // ./lab5eP "3+4+5"



    // windows
    // g++ L4_ejer1.cpp -o L4_ejer1.exe windows
    // .\L4_ejer1.exe input.txt
    delete exp;
}
```