```cpp
// Ejercicio elaborado por el profesor Igor Siveroni //


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

using namespace std;


// Clases //

class Token {
public:
    enum Type {PLUS, MINUS, NUM, ERR, 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 };

class Exp {
public:
    virtual void print() = 0;
    virtual int interprete() = 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 interprete();
    ~BinaryExp();
};

class NumberExp : public Exp {
public:
    int value;
    NumberExp(int v);
    void print();
    int interprete();
    ~NumberExp();
};

```

```
65  class Parser {
66  private:
67      Scanner* scanner;
68      Token *current, *previous;
69      bool match(Token::Type ttype);
70      bool check(Token::Type ttype);
71      bool advance();
72      bool isAtEnd();
73      Exp* parseExpression();
74      Exp* parseTerm();
75      Exp* parseFactor();
76      bool tokenToOp(Token* tk, BinaryOp& op);
77  public:
78      Parser(Scanner* scanner);
79      Exp* parse();
80  };
81
82
83  Token::Token(Type type):type(type) { text = ""; }
84
85  Token::Token(Type type, char c):type(type) { text = c; }
86
87  Token::Token(Type type, const string& source, int first, int last):type(type) {
88      text = source.substr(first,last);
89  }
91  std::ostream& operator << ( std::ostream& outs, const Token & tok )
92  {
93      if (tok.text.empty())
94          return outs << tok.type;
95      else
96          return outs << "TOK" << "(" << tok.text << ")";
97  }
98
99  std::ostream& operator << ( std::ostream& outs, const Token* tok ) {
100     return outs << *tok;
101 }
102
103 // SCANNER //
104
105 Scanner::Scanner(const char* s):input(s),first(0), current(0) { }
107 Token* Scanner::nextToken() {
108     Token* token;
109     while (input[current]==' ') current++;
110     if (input[current] == '\0') return new Token(Token::END);
111     char c  = input[current];
112     first = current;
113     if (isdigit(c)) {
114         current++;
115         while (isdigit(input[current]))
116             current++;
117         token = new Token(Token::NUM,input,first,current-first);
118     } else if (strchr("+-", c)) {
119         switch(c) {
120             case '+': token = new Token(Token::PLUS,c); break;
121             case '-': token = new Token(Token::MINUS,c); break;
122             default: cout << "No deberia llegar aca" << endl;
123         }
124         current++;
125     } else {
126         token = new Token(Token::ERR, c);
127         current++;
128     }
129     return token;
130 }
```

```
132  Scanner::~Scanner() { }
133
134  // PARSER //
135
136  bool Parser::match(Token::Type ttype) {
137      if (check(ttype)) {
138          advance();
139          return true;
140      }
141      return false;
142  }
143
144  bool Parser::check(Token::Type ttype) {
145      if (isAtEnd()) return false;
146      return current->type == ttype;
147  }
148
149  bool Parser::advance() {
150      if (!isAtEnd()) {
151          Token* temp =current;
152          if (previous) delete previous;
153          current = scanner->nextToken();
154          previous = temp;
155          if (check(Token::ERR)) {
156              cout << "Parse error, unrecognised character: " << current->text << endl;
157              exit(0);
158          }
159          return true;
160      }
161      return false;
162  }
163
164  bool Parser::isAtEnd() {
165      return (current->type == Token::END);
166  }
167
168  Parser::Parser(Scanner* sc):scanner(sc) {
169      previous = current = NULL;
170      return;
171  };
172
173  Exp* Parser::parse() {
174      current = scanner->nextToken();
175      if (check(Token::ERR)) {
176          cout << "Error en scanner - caracter invalido" << endl;
177          exit(0);
178      }
179      Exp* exp = parseExpression();
180      if (current) delete current;
181      return exp;
182  }
183
184  Exp* Parser::parseExpression() {
185
186      Exp* left = parseTerm();
187
188      while (match(Token::PLUS) || match(Token::MINUS)) {
189          BinaryOp op = (previous->type == Token::PLUS) ? PLUS : MINUS;
190          Exp* right = parseTerm();
191          left = new BinaryExp(left, right, op);
192      }
193
194      return left;
195  }
196
```

```cpp
Exp* Parser::parseTerm() {
    Exp* e = parseFactor();
    return e;
}

Exp* Parser::parseFactor() {
    if (match(Token::NUM)) {
        return new NumberExp(stoi(previous->text));
    }
    cout << "Error: se esperaba un número." << endl;
    exit(0);
}

bool Parser::tokenToOp(Token* tk, BinaryOp& op)  {
    switch(tk->type) {
        case Token::PLUS: op = PLUS; break;
        case Token::MINUS: op = MINUS; break;
        default: cout << "Invalid Operator" << endl; return false;
    }
    return true;
}

char Exp::binopToChar(BinaryOp op) {
    char  c=' ';
    switch(op) {
        case PLUS: c = '+'; break;
        case MINUS: 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::interprete() {
    int result;
    switch(this->op) {
        case PLUS: result = 0; break;
        case MINUS: result = 0; break;
        default:
            cout << "Operador desconocido" << endl;
            result = 0;
    }
    return result;
}

```

```cpp
int NumberExp::interprete() {
    return 0;
}


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 << "interprete: ";
    cout << exp->interprete() << endl;

    delete exp;
}
```