POO - PROYECTO 3

Claudia Noche

PARTES DE LA CALCULADORA

PARSER

Analizador sintáctico
Recursive Descent Parser (top down)

LEXER

Creación de Tokens y sus partes (Kind, Token, Token_stream)

ERROR

Manejo de Errores y contador

TABLE

Estructura que almacena variables y constantes en la forma <String, Token>

LEXER

KIND

Clase enumerada de caracteres. Estos conforman los distintos símbolos utilizados por la calculadora

```
enum class Kind : char {
    NAME, NUMBER, END,
    PLUS = '+',
    SUB = '-',
    MULT = '*',
    DIV = '/',
    PRINT = ';',
    ASSIGN = '=',
    LP = '(',
    RP = ')',
    EXP = '^'
};
```

```
struct Token {
    Kind kind;
    std::string string_value;
    double number_value;
};
```

TOKEN

Estructura que tiene tanto un valor, ya sea numérico o string, y un tipo (Kind) que sale de la clase Kind

LEXER::TOKEN_STREAM

TOKEN_STREAM

Clase que se encarga de “tokenizar” un input, revisando caracter por caracter su contenido.

```
class Token_stream {
    public:
        Token_stream(std::istream& s) : ip{&s}, owns{false}, ct{Kind::END} { }
        Token_stream(std::istream* p) : ip{p}, owns{true}, ct{Kind::END} { }

        ~Token_stream() { close(); }

        Token get(){ // read and return next token
```

Se caracteriza por tener:

istream *ip - Puntero al input

bool owns - Pertenencia del istream

Token ct - Token actual

```
private:
    void close() { if (owns) delete ip; }

    std::istream* ip; // pointer to an input stream
    bool owns; // does the Token_stream own the istream?

    Token ct {Kind::END}; // current token
```

Al finalizar, se extrae una instancia del objeto

```
extern Lexer::Token_stream ts;
```

LEXER::TOKEN_STREAM

Dos métodos principales:

get() - devuelve el siguiente token

current () - devuelve el token actual

```
Token get(){ // read and return next token
    char ch = 0;
    do{
        if(!ip->get(ch))
        return ct={Kind::END};
    }
    while (ch!='\n' && isspace(ch));
    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::PRINT};  // end of expression; print
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '^':
    case '=':
        return ct = {static_cast<Kind>(ch)};
    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
    case '.':
        ip -> putback(ch); // put the first digit (or .) back into the input stream
        *ip >> ct.number_value; // read number into ct
        ct.kind = Kind::NUMBER;
        return ct;
    default: // name, name =, or error
        if (isalpha(ch)) {
            ct.string_value = ch;
            while (ip->get(ch) && isalnum(ch))
                ct.string_value += ch; // append ch to end of string_value
                ip->putback(ch);
                return ct={Kind::NAME};
        }
    error("bad token");
    return ct={Kind::PRINT};
```

```
Token& current(){ // most recently read token
    return ct;
}
```

PARSER

EXPR

Maneja sumas y restas

```
double expr(bool get) { //add y sub
    double left = term(get);
    while(true) {
        switch (ts.current().kind) {
        case Kind::PLUS:
            left += term(true);
            break;

        case Kind::SUB:
            left -= term(true);
            break;
            default:
            return left;
        }
    }
}
```

TERM

Maneja multiplicación y división

```
double term(bool get) { //mult y div y exp
    double left = prim(get);
    while(true) {
        switch (ts.current().kind) {
        case Kind::MULT:
            left *= prim(true);
            break;
        case Kind::DIV:
            if (auto d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        case Kind::EXP:
            left = pow(left, prim(true));
            break;
        default:
            return left;
        }
    }
}
```

MORE PARSER

PRIM

Maneja expresiones

```
double prim(bool get) {
    if (get) ts.get(); // read next token
        switch (ts.current().kind) {
        case Kind::NUMBER: { // floating-point constant
            double v = ts.current().number_value;
            ts.get();
            return v;

        }
        case Kind::NAME: {
            double& v = Table::table[ts.current().string_value];
            if (ts.get().kind == Kind::ASSIGN)
                v = expr(true); // '=' seen: assignment
                return v;
        }
        case Kind::SUB: // suma y minus
            return -prim(true);
        case Kind::LP: {
            auto e = expr(true);
            if (ts.current().kind != Kind::RP)
                return error("')' expected");
                ts.get(); // eat ')'
                return e;
        }
    default:
        return error("primary expected");
    }
}
```

ERRORES

```
int no_of_errors;
double error(const std::string& s) {
    ++no_of_errors;
    std::cerr << "Error: " << s << '\n';
    return 1;
}
```

Los errores propios de la calculadora se manejan con la creación de función que imprime un mensaje previamente establecido si encuentra un error.

Asimismo, se crea una variable que cuenta la cantidad de errores, llamada **no_of_errors**

Ejemplo::**Prim**

```
        case Kind::LP: {
            auto e = expr(true);
            if (ts.current().kind != Kind::RP)
                return error("')' expected");
                ts.get(); // eat ')'
                return e;
        }
    default:
        return error("primary expected");
    }
```

TABLA

Se implementa un map de strings y doubles que guarda tanto constantes (implementadas dentro del código, por ejemplo) como variables creadas en el momento.

```
namespace Table{
    std::map<std::string,double> table;
}
#endif
```

VALIDACIONES EN EL MAIN

Finalmente, se instancia el objeto **Token_stream** con un input.

```
Lexer::Token_stream ts {&cin};

void calculate(){
    while(true){
        ts.get();
        if (ts.current().kind == Lexer::Kind::END) break;
        if (ts.current().kind == Lexer::Kind::PRINT) continue;
        std::cout << expr(false) << '\n';
    }
}
```

```
int main(int argc, char* argv[]) {
    istream* input;
    switch (argc) {
        case 1: // read from standard input
            input=&cin;
            break;
        case 2: // read from argument string
            ts.set_input(new istringstream{argv[1]});
            break;
        default:
            error("too many arguments");
            return 1;
    }
    Table::table["pi"] = 3.1415926535897932385; // inser t predefined names
    Table::table["e"] = 2.7182818284590452354;
    calculate();
    cout<<endl;
    return no_of_errors;
}
```

Se valida que se tenga un número de argumentos válidos

Se crean dos constantes y se llama la función calculate

CONCLUSIONES

El uso de un parser recursivo de tipo Top-Down asegura simplicidad en el código, aunque tome más tiempo de finalizar. Se presenta como la opción más eficiente, pues es un programa relativamente pequeño.

La separación de componentes en un programa permite encontrar errores de forma más eficiente.