Proyecto 3

Eduardo Medina

Desktop calculator

Dividida en 4 partes:

- Parser: Análisis sintáctico.
- Input: Análisis léxico e input.
- Symbol table: Se usa para información permanente.
- Driver: Maneja la inicialización, output y errores.

Lo más importante de cada parte:

Parser

```
double term(bool);
double expr(bool get){...}
double prim(bool get){...}
double term(bool get){...}
```

Se declara de esta forma porque existen dependencias.

Parser

Parser usa Token_stream para encapsular la lectura de caracteres y su composición en Tokens.

De la clase Token_stream, el parser usa:

- Para leer el siguiente Token llama a Ts.get().
- Para leer el token más reciente llama a ts.current().

Cada función recibe un booleano, llamado get(), indicando si la función debe llamar a Token_stream::get().

Parser / expr

```
double expr(bool get){
    double left = term(get);

    for(;;){
        switch(ts.current().kind){
            case Kind::plus:
                left += term( get: true);
                break;
            case Kind::minus:
                left -= term( get: true);
                break;
            default:
                return left;
        }
    }
}
```

Llama a otra función para que haga el trabajo.

Parser / term

```
double term(bool get){
    double left = prim(get);

    for(;;){
        switch (ts.current().kind){
            case Kind::mul:
                left *= prim( get: true);
                break;
            case Kind::div:
                if(auto d = prim( get: true)){
                    left /= d;
                    break;
                }
                return error( s: "divide by 0");
            default:
                return left;
        }
    }
}
```

Si recibe cero retorna error. Esto lo evalúa en prim(true). Si recibe cero, esto será false y pasará de frente al error.

Llama a otra función para que haga el trabajo.

Parser / prim

```
double prim(bool get){
    if(get) ts.get();

    switch(ts.current().kind){
        case Kind::number:
        {
            double v = ts.current().number_value;
            ts.get();
            return v;
        }
        case Kind::name:
        {
            double& v = table[ts.current().string_value];
            if(ts.get().kind == Kind::assign) v = expr( get: true);
            return v;
        }
        case Kind::minus:
            return -prim( get: true);
        case Kind::lp:
        {
            auto e = expr( get: true);
            if(ts.current().kind != Kind::rp) return error( s: "')' expectend");
            ts.get();
            return e;
        }
        default:
            return error( s: "primary expected");
    }
}
```

- Cuando un token que es un número es visto, su valor es colocado en number_value.
-
- Cuando es un string es colocado en string_value.

Lexer

```
enum class Kind : char{
    name, number, end,
    plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};
struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

Para el input el parser utiliza Token_stream, que encapsula la lectura de caracteres y su composicion en Tokens.

Los tokens son de tipo {kind-of-token,value}

Lexer / Token_stream

```
class Token_stream {
public:
    explicit Token_stream(istream& s) : ip{&s},owns(false), ct{ .kind: Kind::end}{}
    explicit Token_stream(istream* p) : ip{p},owns(true), ct{ .kind: Kind::end}{}
    ~Token_stream(){close();}

    Token get(){
        char ch;
        do{
            if(!ip->get( &: ch)) return ct={ .kind: Kind::end};
        } while(ch!='\n' && isspace(ch));

        switch(ch){
            case ';':
            case '\n':
                return ct={ .kind: Kind::print};
            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);
                *ip >> ct.number_value;
                ct.kind=Kind::number;
                return ct;
```

Token_stream “owns” an istream passed as pointer but not an instream passed as reference. Útil para clases que contienen recursos que requieren destruirse.

get() -> Lee un carácter, usarlo para saber que Token componer, leer más Tokens, si es necesario y retornar un Token.

Lexer / Token_stream

```
            default:
                if(isalpha(ch)){
                    ct.string_value = ch;
                    while(ip->get( &: ch)){
                        if(isalnum(ch))
                            ct.string_value += ch;
                        else{
                            ip->putback(ch);
                            break;
                        }
                    }
                    ct.kind = Kind::name;
                    return ct;
                }
                error( s: "bad token");
                return ct={ .kind: Kind::print};
        }
    }
    Token & current(){
        return ct;
    }
    void set_input(istream& s) {close(); ip=&s; owns=false;}
    void set_input(istream* p) {close(); ip=p; owns=true;}
private:
    void close() {if (owns) delete ip;}
    istream* ip;
    bool owns;
    Token ct{ .kind: Kind::end};
};
extern Token_stream ts;
```

Error Handling

```
#ifndef DESKCALCULATOR_ERROR_H
#define DESKCALCULATOR_ERROR_H
#include <iostream>
#include <sstream>

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

Cerr -> Unbuffered output stream usually used to report errors.

Symbol table

```
extern map<string,double> table;
```

Guarda valores fijos, como los que se verán en el main.

Driver

```
#include "input.h"

int no_of_errors;
map<string,double> table;
Token_stream ts{ &: cin};

void calculate(){
    for (;;) {
        ts.get();
        if (ts.current().kind == Kind::end) break;
        if (ts.current().kind == Kind::print)continue;
        cout << expr( get: false) << '\n';
    }
};

int main(int argc, char* argv[]){
    table["pi"]=3.14159265;
    table["e"]=2.718281828;
    calculate();
    return no_of_errors;
}
```

cout << expr(false) (IMPRIME LA RESPUESTA)

False le dice a expr que no necesita llamar a ts.get() para leer un token.

end -> el loop se termina cuando se encuentra un error o eof.

Argc y argv transmiten argumentos del programa. Argc especifica la cantidad de argumentos y argc, el array de argumentos

Conclusiones

- La implementación de una calculadora no suena muy complicado, pero al ver la estructura y desarrollo de este compilador queda claro que aún nos queda mucho por aprender y que existen herramientas mucho más interesantes de las que utilizamos para programar en nuestro día a día.