`cppcalc` en C++

Edison Valencia Díaz
José Luis Montoya Pareja
Juan Francisco Cardona McCormick

Fecha de entrega: 14 de Noviembre de 2016

1. Introducción

Durante una parte del semestre hemos aprendido programación orientada a objetos con el lenguaje de programación C++ construyendo un interpretador de expresiones con memoria. Ésta práctica, continuará ampliando este proyecto para aprender otros aspectos del lenguaje de programación C++, como entrada y salida, colecciones y otros. En este documento se propondrán los cambios que se harán para el proyecto `cppcalc`.

2. Definición

Son varias las modificaciones que vamos hacer al proyecto:

- La gramática será una gramática de instrucciones, con la instrucción de asignación como la instrucción preferencial.
- El programa permitirá inicializar variables desde las variables de ambiente y línea de comandos.
- Ampliación de las funciones de la memoria de la calculadora.
- `cppcalc` en un generador de código EWE.

Aunque son varias modificaciones al proyecto original, esto no implica que vamos a tener varios programas: uno para la calculador original, otro

para las lectura interactiva de varias expresiones, otro para el generador de código EWE, etc. ***Se va a generar un único ejecutable*** `cppcalc`[1] (mirar las secciones: 2.1.5, 2.1.2 y 2.2).

2.1. Gramática modificada

La siguiente es la nueva versión de la gramática independiente de contexto $LL(1)$:

$$
\begin{array}{lcl}
Prog & \rightarrow & Stms \text{ EOF} \\
Stmts & \rightarrow & \textbf{identifier} = Expr \text{ ; } EOL\ Stmts \mid EOL \mid \epsilon \\
Expr & \rightarrow & Term\ RestExpr \\
RestExpr & \rightarrow & + Term\ RestExpr \mid \text{- } Term\ RestExpr \mid \epsilon \\
Term & \rightarrow & Storable\ RestTerm \\
RestTerm & \rightarrow & * Term\ RestTerm \mid / Term\ RestTerm \mid \% Term\ RestTerm \mid \epsilon \\
Storable & \rightarrow & Negation\ MemOperation \\
MemOperation & \rightarrow & \text{S} \mid \text{P} \mid \text{M} \mid \epsilon \\
Negation & \rightarrow & \text{- } Factor \mid Factor \\
Factor & \rightarrow & \textbf{number} \mid \textbf{identifier} \mid \text{R} \mid \text{C} \mid (\ Expr\ )
\end{array}
$$

Esta gramática permite manejar una lista de expresiones:

```
$ ./cppcalc
> a = 3 + 4;<eol>
= a <- 7
> b = 8 + 9;<eol>
= b <- 17
> <eol>
=
> c = 8 + <eol>
* parse error line: 4 col: 8
> a = 3 - 2;<eol>
= a <- 1
> <eof>
$
```

[1] No es necesario poner la extensión `.exe` por que se sobrentiende que es un ejecutable del lenguaje de C++.

Se observa que el programa ya no envía el mensaje de solicitar una expresión, sino que muestra lo que se conoce como un *prompt*[2]. Este ***prompt*** indica al usuario que entre una expresión y el programa responderá con = para indicar el resultado o $*$ para indicar un error. El usuario debe entrar una sesión con una línea nueva < `eol` >, esta línea nueva es simplemente presionar la tecla *return*[3]. Finalmente, el usuario puede dejar de entrar expresiones al indicar un fin de fichero (*eof - end of file*). El usuario puede indicar el fin de un fichero utilizando en Linux (Unix o Mac) la combinación de las teclas `Ctrl+D`. Con el fin de fichero el programa `cppcalc` termina su ejecución y vuelve el ***prompt*** de la terminal para esperar otro comando.

2.1.1. Funciones de memoria de la calculadora

La calculadora contaba con dos instrucciones de memoria: *Store* `S` y *Recall* (`R`). Ahora la calculadora contará con más instrucciones para manejar la memoria: *Plus* (`P`) para adicionar a la memoria el resultado del sub-árbol $M = M + eval(Sub_{tree})$; *Minus* (`M`) para restar a la memoria el resultado del sub-árbol $M = M - eval(Sub_{tree})$; *Clear* (`C`) para establecer en 0 el valor de la memoria $M = 0$.

2.1.2. Sintaxis línea de comandos

La siguiente es la sintaxis de la línea de comandos:

$$
\begin{array}{lcl}
Comandos & \rightarrow & [InitVar] \ \ldots \ [Gen] \ [Fichero] \ \ldots \\
InitVar & \rightarrow & \text{-v "}\textbf{identifier} = \text{number"} \\
Gen & \rightarrow & \text{-g} \\
Fichero & \rightarrow & \textbf{filename}
\end{array}
$$

Una línea de comando tiene una lista posiblemente vacía de los valores de los identificadores, cada inicialización asigna a una variable un valor entero, si una variable está repetida en la lista, el último valor asignado es el valor de la variable; luego sigue una opción que indica si la calculadora genera código;

[2]Muchas de las expresiones que vamos a utilizar son dadas en inglés por que en español no existe una traducción adecuada. En este caso *prompt* tiene traducción literal en inglés de solicitar a alguien de decir o hacer algo, en este caso > es una forma que la calculadora le indica al usuario que digite una expresión.

[3]Esto depende de la plataforma en Linux, Unix, Mac y Cygwin/Windows es simplemente un sólo carácter \n, en Windows es la combinación de dos caracteres \r\n. El programa debe ser independiente de la plataforma.

luego sigue una lista de ficheros posiblente vacía donde tiene las expresiones a evaluar.

2.1.3. Variables de ambiente

Los valores iniciales de una variable puede ser inicializados utilizando las variables de ambiente de la siguiente forma:

```
$ export CALCVAR_b=10
$ ./cppcalc
> a = 3 + b;<eol>
= a <- 13
> <eof>
$
```

2.1.4. Inicialización de variables

Las variables (o identificadores) de una expresión de consideran que tiene valores cero:

```
$ ./cppcalc
> a = 3 + b;<eol>
= a <- 3
> <eof>
$
```

La gramática permitirá la inicialización de variables, a través de la línea de comandos:

```
$ export CALCVAR_b=10
$ export CALCVAR_c=20
$ ./cppcalc
> a = 3 + b;<eol>
= a <- 13
> a = 2 * c;<eol>
= a <- 40
> <eof>
$
```

También a través de variables de ambiente:

```
$ ./cppcalc -v b=10 -v c=20
> a = 3 + b;<eol>
= a <- 13
> a = 2 * c;<eol>
= a <- 40
> <eof>
$
```

O ambos métodos, en cuyo caso tiene preferencia las variables definidas en la línea de comandos:

```
$ export CALCVAR_b=10
$ export CALCVAR_c=30
$ ./cppcalc -v b=10 -v c=20
> a = 3 + b;<eol>
= a <- 13
> a = 2 * c;<eol>
= a <- 40
> <eof>
$
```

2.1.5. Lectura de más de un fichero

La entrada de las expresiones se puede cambiar si estas son leídas directamente de uno o más ficheros. Suponga que tenemos dos ficheros con extensión `.calc` para indicar que son programas que contiene expresiones aritméticas:

```
$ cat expresion1.calc
a = 1 + b;
b = 2 * 4S + R;
c = (3 + 4) * d;
<eof>
```

```
$ cat expresion2.calc
b = 1S + 4M + R;
b = a + c + d;
<eof>
```

Para ejecutar `calc` con fichero en la línea de comando se indica al final el nombre del fichero que se quiere procesar:

```
$ ./cppcalc expresion1.calc
= a <- 1
= b <- 12
= c <- 0
```

Se puede procesar con más de un fichero:

```
$ ./cppcalc expresion1.calc expresion2.calc
= a <- 1
= b <- 12
= c <- 0
= b <- -5
= b <- 1
```

También se puede combinar con la inicialización de la variables:

```
$ ./cppcalc -v a=1 -v b=2 -v c=3 -v d=4 expresion1.calc expresion2.calc
= a <- 3
= b <- 12
= c <- 28
= b <- -5
= b <- 35
```

2.2. `cppcalc` es un generador de código

La calculadora fue diseñada como un interpretador, es decir que la calculadora lee una expresion e inmediamente evalua la expresión. Vamos a modificar el proyecto para que `calc` sea un generador de código para EWE, en vez de ser un interpretador.

A continuación mostramos una ejecución de `calc` como generador de código:

```
$ ./cppcalc -g
> a = 3S * 4 + R;<eol>
> <eof>
$
```

En la anterior interacción: `cppcalc` recibe una línea de comando `-c` (ver 2.1.2) que indica a `cppcalc` que va a generar un fichero llamado `a.ewe`[4] que contiene lo siguiente[5]:

[4]Siempre que no se indique un fichero de entrada en cuyo caso el nombre del fichero de salida sera el mismo del fichero de entrada.

[5]`cat` es un programa de Linux y cygwin que permite ver el contenido de un fichero

```
$ cat a.ewe
# Expresion a = 3S * 4 + R;
start:
# Instrucciones antes del recorrido del arbol abstracto sintactico
   sp      := 1000
   one     := 1
   zero    := 0
   memory  := zero
#  Comienza el recorrido del arbol
#  push(3)
   sp      := sp - one
   operator1 := 3
   M[sp+0] := operator1
#  Store
   memory  := M[sp+0]
#  push(4)
   sp      := sp - one
   operator1 := 4
   M[sp+0] := operator1
#  Times
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 * operator2
   sp := sp + one
   M[sp+0]   := operator1
#  Recall
   sp      := sp - one
   M[sp+0] := memory
#  Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
#  Assign
   a := M[sp+0]
#  Write result
   operator1 := M[sp+0]
```

```
   sp := sp - one
   writeInt(operator1)
end: halt
equ  memory       M[0]
equ  one          M[1]
equ  zero         M[2]
equ  operator1    M[3]
equ  operator2    M[4]
equ  sp           M[5]
equ  a            M[6]
equ  stack        M[1000]
```

Suponga que ahora la interacción con `cppcalc` es la siguiente:

```
$ ./cppcalc -c
> a = 2S + (3S + R)M + C;<eol>
> <eof>
$
```

El fichero de salida `a.ewe` es el siguiente:

```
$ cat a.ewe
# Expresion: a = 2S + (3S + R)M + C;
start:
# Instrucciones antes del recorrido del arbol abstracto sintactico
   sp      := 1000
   one     := 1
   zero    := 0
   memory  := zero
# Comienza el recorrido del arbol en postorden
# push(2)
   sp := sp - one
   operator1 := 2
   M[sp+0] := operator1
# Store
  memory := M[sp+0]
#  push(3)
   sp := sp - one
```

```
   operator1 := 3
   M[sp+0] := operator1
# Store
   memory := M[sp+0]
# Recall
   sp := sp - one
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
# Memory Minus
   operator2 := M[sp+0]
   memory := memory - operator2
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
# Clear
   memory := zero
   sp := sp - one
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
# Assign
   a := M[sp+0]
# Write Result
   operator1 := M[sp+0]
   sp := sp - one
```

```
   writeInt(operator1)
end: halt
equ  memory       M[0]
equ  one          M[1]
equ  zero         M[2]
equ  operator1    M[3]
equ  operator2    M[4]
equ  sp           M[5]
equ  a            M[6]
equ  stack        M[1000]
```

2.2.1. Generación de código para más de una expresión

Como la iteracción con la calculadora es de más de una expresión se puede dar la siguiente interacción:

```
$ ./cppcalc -g
> a = 3S * 4 + R;<eol>
> b = 2S + (3S + R)M + C;<eol>
> <eof>
$
```

El archivo obtenido `a.ewe` es el siguiente [6]:

```
$ cat a.ewe
# Expresion a = 3S * 4 + R;
expr1:
# Instrucciones antes del recorrido del arbol primer expresion
   sp      := 1000
   one     := 1
   zero    := 0
   memory  := zero
#  Comienza el recorrido del arbol
#  push(3)
   sp      := sp - one
   operator1 := 3
   M[sp+0] := operator1
```

[6] Tenga en cuenta que entre cada asignación el valor de la memoria se conserva

```
#  Store
   memory  := M[sp+0]
#  push(4)
   sp      := sp - one
   operator1 := 4
   M[sp+0] := operator1
#  Times
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 * operator2
   sp := sp + one
   M[sp+0]   := operator1
#  Recall
   sp      := sp - one
   M[sp+0] := memory
#  Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
#  Assign
   a := M[sp+0]
#  Write result
   operator1 := M[sp+0]
   sp := sp - one
   writeInt(operator1)
# Expresion: b = 2S + (3S + R)M + C;
expr2:
# Instrucciones antes del recorrido de la segunda expresion
   sp      := 1000
   one     := 1
   zero    := 0
# La memoria se conserva entre expresión y expresión
# Comienza el recorrido del arbol en postorden
# push(2)
   sp := sp - one
   operator1 := 2
```

```
   M[sp+0] := operator1
# Store
  memory := M[sp+0]
#  push(3)
   sp := sp - one
   operator1 := 3
   M[sp+0] := operator1
# Store
   memory := M[sp+0]
# Recall
   sp := sp - one
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
# Memory Minus
   operator2 := M[sp+0]
   memory := memory - operator2
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
# Clear
   memory := zero
   sp := sp - one
   M[sp+0] := memory
# Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0] := operator1
```

```
# Assign
   b := M[sp+0]
# Write Result
   operator1 := M[sp+0]
   sp := sp - one
   writeInt(operator1)
end: halt
equ  memory       M[0]
equ  one          M[1]
equ  zero         M[2]
equ  operator1    M[3]
equ  operator2    M[4]
equ  sp           M[5]
equ  a            M[6]
equ  b            M[7]
equ  stack        M[1000]
```

2.2.2. Generación de código con inicialización de variables

La compilación a ewe también acepta la inicialización de variables:

```
$ ./cppcalc -v a=2 -v b=12 -g
> c = (a + 10) * b;<eol>
> <eof>
$
```

Producirá el siguiente código:

```
$ cat a.ewe
# Expresion c = (a + 10) * b;
start:
# Instrucciones antes del recorrido del arbol abstracto sintactico
   sp      := 1000
   one     := 1
   zero    := 0
   a       := 2
   b       := 12
   memory  := zero
#  Comienza el recorrido del arbol
```

```
#  push(a)
   sp      := sp - one
   operator1 := a
   M[sp+0] := operator1
#  push(10)
   sp      := sp - one
   operator1 := 10
   M[sp+0] := operator1
#  Add
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 + operator2
   sp := sp + one
   M[sp+0]   := operator1
#  push(b)
   sp      := sp - one
   operator1 := b
   M[sp+0] := operator1
#  Times
   operator2 := M[sp+0]
   operator1 := M[sp+1]
   operator1 := operator1 * operator2
   sp := sp + one
   M[sp+0] := operator1
#  Assign
   c := M[sp+0]
#  Write result
   operator1 := M[sp+0]
   sp := sp - one
   writeInt(operator1)
end: halt
equ  memory       M[0]
equ  one          M[1]
equ  zero         M[2]
equ  operator1    M[3]
equ  operator2    M[4]
equ  sp           M[5]
equ  a            M[6]
```

```
equ  b            M[7]
equ  c            M[8]
equ  stack        M[1000]
```

2.2.3. Generación de código para más de un fichero

La calculadora también permite generar más de un fichero de código de ewe a partir de ficheros de expresiones tipo `.calc`. Tomando por ejemplo los siguientes ficheros:

```
$ cat  expresion3.calc          $ cat  expresion4.calc
a = 3 + 4;                      b = 3 * 4;
```

Se puede compilar todos al tiempo utilizando la siguiente opción del generador de código:

```
$ ./cppcalc -g expresion3.calc expresion4.calc
```

2.2.4. Ejecución de programas generados

Todos los programas generados para ewe no deben tener errores de sintaxis en ewe y deben ser permitir ser ejecutados, por ejemplo el resultado obtenido en la sección 2.2.2 puede ser ejecutado sin problemas en ewe, recuerde que este código ya tiene las variables definidas inicializadas:

```
$ ewe a.ewe
114
```

También cuando se generar más de un fichero estos pueden ejecutarse directamente con ewe, por ejemplo con los generados en la sección 2.2.3:

```
$ ewe expresion3.ewe expresion4.ewe
7
12
```

2.2.5. Ejemplo de manejo de mapas para ambientes

En la práctica es necesario utilizar colecciones para poder almacenar y manipular las variables. En este caso vamos a utilizar los mapas (también llamados diccionarios) que permiten almacenar información en forma de clave y su valor. En el siguiente ejemplo mostramos como se utilizan los mapas para obtener una lista de valores y mostrar su frecuencia:

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int
main(void) {
  map<string, int> env;
  string var;
  while (cin >> var) {
    env[var]++;
  }

  for (map<string, int>::iterator it = env.begin();
       it != env.end(); ++it) {
    cout << it->first << " = " << it->second << endl;
  }

  return 0;
}
```

El anterior programa al ejecutar con un conjunto valores produce el siguiente resultado:

```
$ ./environment
hola
mundo
adios
mundo
hola
adios
```

```
prueba
<eof>
adios = 2
hola = 2
mundo = 2
prueba = 1
```

3. Requerimientos técnicos

Directorios La siguiente es la estructura de directorios del proyecto

```
+-+
  |-+ proyectos +
                |-+ cppcalc
                |-+ cppcalc-examples
```

Lenguaje de programación: Esta primera práctica debe ser hecha en el lenguaje de programación C++.

Manejo de proyectos: Para manejar proyectos se debe utilizar la herramienta *make*.

Control de versiones: Se utilizará el manejador de versiones *subversion*.

Repositorio: El proyecto debe estar alojado en `http://riouxsvn.com` como se ha venido trabajando en clase.

4. Requerimientos administrativos

Trabajo grupal: El trabajo para esta práctica es posible que sea hasta grupos de dos personas. *Cada* estudiante está registrardo en (Riouxsvn) con su usuario. Para el estudiante individual mantedrá su repositorio actual. Para los grupos de 2 personas, el grupo debe crear un repositorio nuevo en Riouxsvn con un nombre que tenga el prefijo $244g$ y un sufijo con un nombre único en minúsculas máximo de carácteres, por ejemplo: $244gjazzrock$. Ambos integrantes deben tener permisos de dueños y deben invitar a su profesor correspondiente.

Entrega: 14 de Noviembre hasta las 18:00 horas. Ésta se hará automática a través del sistema de control de versiones. Se *debe* cumplir todos los requerimientos señalados en este documento.

Sustentación: El mismo se enviará un correo con un enlace donde se pueden registrar para hacer las sustentaciones.

Pruebas: El proyecto se copiará con una serie de pruebas que consiste en un conjunto de expresiones: 20 en total. Cada expresion se probará, cada una con cada modalidad del programa:

- Sin cambios.
- Ejecuciones con más de un fichero.
- Ejecuciones con inicialización de variables (línea de comandos y variables de ambiente).
- Generador de código a EWE.
- Generador de código a EWE con inicialización de variables (línea de comandos y variables de ambiente).

Una semana antes de la entrega de la práctica se entregará un enlace donde pueden encontrar un 40 % de las pruebas que serán llevadas a cabo en la sustentación para su propia prueba personal.

Nota práctica: Se calcula con la siguiente formula.

$$Nota\ final = Sustentación \times (5 \times \sum_{j=1}^{5}(\frac{1}{5} * (\frac{(\sum_{i=1}^{20} prueba_i^j)}{20})))$$

Donde $Sustentación$ es valor obtenido en la sustentación ($0 \leq Sustentación \leq 1$); $prueba_i^j$ indica si la prueba i pasó o falló en el caso j ($0 \leq prueba_i^j \leq 1$).
En caso que ninguna prueba pase se hará un revisión visual de la práctica revisando el código observando que el código cumpla con los requerimientos establecidos en la práctica. De este revisión visual el estudiante puede obtener una nota entre 0,0 y 2,5 como máximo.

Control de versiones: El control de versiones no es solamente un herramienta que facilite la comunicación entre los miembros del grupo y del

control de versiones, sino que también ayudará al profesor a llevar un control sobre el desarrollo de la práctica. Se espera que las diferentes registros dentro del control de versiones sean cambios graduales. En caso contrario, se procederá a realizar un escutrinio a fondo del manejo de control de versiones para evitar fraudes.

Fraudes: Seguiremos los lineamientos establecidos por la universidad con respecto a las conductas que atentan contra el orden académico. En el suguiente enlace `http://www.eafit.edu.co/institucional/reglamentos/Documents/pregrado/regimen-disciplinario/cap1.pdf` se encuentra la parte del régimen disciplinario de la universidad.