Especificación de la sintaxis del Lenguaje de Programación C+ϵ

Enrique Rey Gisbert y Juan Carlos Llamas Núñez

13 de mayo de 2022

1. Enfoque del Lenguaje de Programación

El objetivo de este documento es presentar la sintaxis del lenguaje de programación que los alumnos Enrique Rey Gisbert y Juan Carlos Llamas Núñez hemos construido para la asignatura de Procesadores del Lenguaje, denominado C+ϵ. Es un lenguaje imperativo guiado por instrucciones secuenciales similares a las vistas en lenguajes como C++ o Java. Sin embargo, debido a la alta complejidad que suele tomar realizar un compilador para un lenguaje de estas características, nos hemos reducido a un subconjunto de tipos, operaciones, instrucciones (que serán suficientes para que el lenguaje sea Turing Completo) y bloques de código, con una sintaxis prefijada, además de introducir ciertas modificaciones en algunas instrucciones típicas que simplifican considerablemente el proceso de diseño e implementación.

En primer lugar, vamos a describir la estructura general que tiene un programa escrito en este lenguaje de programación, así como las estructuras más amplias que engloban el resto del código. En segundo lugar, expondremos las unidades léxicas que necesitamos junto con la sintaxis de los componentes e instrucciones que tiene nuestro lenguaje de programación. Por último, daremos múltiples ejemplos de programas sencillos escritos en el lenguaje C+ϵ, que utilizan todas las instrucciones que están disponibles.

2. Bloques, Estructuras y Funciones

Un programa en nuestro lenguaje está compuesto por la declaración de una secuencia de tipos estructurados (structs), seguida de la declaración (y posible inicialización) de variables globales, y a continuación por la definición e implementación de una serie de funciones. Aunque la sintaxis concreta de las funciones y de las declaraciones (con y sin inicialización) se explicará posteriormente en este documento, cabe destacar que una de estas funciones debe tener *main* como identificador, no retornar ningún valor y no recibir ningún parámetro. Esta función es por la que comienza el hilo de ejecución del programa. No se permite la creación de módulos y su posterior importación, por lo que deben declararse todas las estructuras, variables y funciones en un mismo archivo de texto.

El cuerpo de las funciones está compuesto por un bloque de código, formado por una secuencia de instrucciones. Algunas de estas instrucciones pueden a su vez contener bloques de código, por ejemplo, if o while. Los diferentes bloques de código se delimitan por el uso de llaves y debemos garantizar el posible y correcto uso de las variables según el ámbito de declaración de las mismas. De esta forma, permitimos el anidamiento de bloques que deberá respetar estas propiedades. Sin embargo, no permitimos la declaración anidada de nuevas funciones ni la definición de lambda expresiones, por lo que se deben declarar todas las funciones de los programas de forma secuencial detrás de las declaraciones de las estructuras y las variables globales.

3. Tipos, Identificadores, Operadores e Instrucciones

Una vez descrita la estructura que tienen los programas escritos en nuestro lenguaje de programación, pasamos a detallar los elementos que se utilizan para construir las instrucciones del lenguaje. Empezaremos con los elementos más básicos como las variables y las constantes, llegando hasta la sintaxis de las instrucciones y las funciones.

Tenemos dos conjuntos de constantes que son los **tipos de datos** básicos de nuestro lenguaje: Int = $\mathbb{Z}$ y Bool = {True, False}. Estos, junto al tipo Array y los tipos estructurados que se declaren (que se explican a continuación) son los tipos que toman las variables. Dichas variables están identificadas por cadenas de caracteres (identificadores) que pueden tener dígitos numéricos, siempre y cuando no estén en primera posición. También usamos identificadores para nombrar a las funciones.

Pueden crearse listas en forma de Arrays (de una o varias dimensiones). Imponemos la restricción de que todos los elementos de la lista deben de ser del mismo tipo y, además, la longitud de un Array debe ser indicada en su declaración. Los posibles tipos de un Array son Int, Bool, otro Array de un cierto tipo y tamaño, o un tipo de Struct previamente declarado. Por tanto, algunos ejemplos de instancias de listas son [1, 2, 3, 4], [ ], [[1, 2], [3, 4]] y [True, False]. Sin embargo, [1,True], [1,[1,2]] o [[True], [False, False]] no pueden ser construidos con nuestra sintaxis. Adicionalmente, se pueden crear expresiones que contienen a variables, funciones y constantes utilizando operadores infijos o prefijos con prioridades fijadas caso por caso.

También hemos introducido Structs, que solo pueden crearse antes de la sección de declaración de variables globales. Los campos del Struct no se pueden inicializar dentro de la declaración, pero pueden ser de distinto tipo, siendo necesario especificarlo caso a caso. Además, es posible representar instancias de Structs genéricas (pueden asignarse a Structs asociados a cualquier declaración, siempre y cuando los tipos de sus campos coincidan) con una notación similar a las instancias de listas, pero usando {} para delimitar el inicio y fin del mismo, como se muestra en el siguiente ejemplo:

```
// Declaracion
Struct MiStruct {
        Array<Int, 2> a;
        Bool b;
}

// Instanciacion e Inicializacion
MiStruct s = {[4,5], True};
```

Para los comentarios que se inserten dentro del código, se indica con // al inicio de los mismos (hasta que haya un salto de línea, momento en el cual se terminará el comentario). También se pueden poner comentarios entre /* y */, sin anidamiento.

Procedemos a introducir algunos **operadores básicos** que contiene nuestro lenguaje. Los operadores asociados a los valores de tipo Int son: + , * , /, % y $\wedge$, que realizan las operaciones de suma, producto, división entera, módulo y potencia. Además, el operador - está sobrecargado y puede usarse como operador binario, representando la resta de dos enteros, o como operador unario prefijo, representando el opuesto de un entero. Todos ellos son operadores (funciones) de tipo Int $\times$ Int $\longrightarrow$ Int, por lo que para definir el operador potencia, para el cuál WebAssembly no proporciona una implementación directa, hemos establecido que su semántica será: a $\wedge$ b = a*...*a (b veces) si b es un entero positivo y a $\wedge$ b = 1 si b es un entero negativo o 0.

Para los booleanos también contamos con algunos operadores básicos: && , || y !, que representan la conjunción, disyunción y negación lógica.

Sobre los Arrays tenemos la operación de acceso a un elemento [ ]. Por tanto, dado la instancia a = [1,2,3,4,5] de Array <Int, 5> se tiene que a[0] devuelve 1, a[1] devuelve 2 y así hasta a[4] que devuelve 5.

Por último, sobre los Structs introducimos también el operador . para acceder a sus campos, que es un operador binario asociativo a izquierdas con prioridad máxima.

Todos los tipos de datos anteriormente descritos tienen otros dos operadores binarios de igualdad o desigualdad que devuelven un valor de tipo Bool. Los operadores son: == y !=. De igual manera, para los enteros tenemos los operadores de desigualdad: <=, >=, < y >. Todos ellos devuelven un valor booleano.

A modo resumen y para especificar correctamente la precedencia, mostramos en la Figura 1 los distintos **niveles de prioridad** y los operadores que se encuentran en cada nivel. Para determinar los nivel de precedencia hemos utilizado la precedencia estándar de otros lenguajes de programación como C. Naturalmente, la precedencia descrita se puede romper con el uso adecuado de paréntesis.

Nivel de prioridad	Operadores	Binario/Unario	Asociatividad
0	.	Binarios	Asociativos a izquierdas
1	! , - , * , &	Unarios	No asociativos
2	$\wedge$	Binario	No asociativo
3	* , / , %	Binarios	Asociativos a izquierdas
4	+ , -	Binarios	Asociativos a izquierdas
5	<= , >= , < , >	Binarios	No asociativos
6	== , !=	Binarios	Asociativos a izquierdas
7	&&	Binario	Asociativo a izquierdas
8	||	Binario	Asociativo a izquierdas

Figura 1: Precedencia de operadores (A menor nivel de prioridad, mayor precedencia)

Presentamos a continuación las **instrucciones** de las que dispone nuestro lenguaje de programación. Junto con la descripción semántica en alto nivel de cada instrucción, se aporta un ejemplo de uso para cada una de ellas, que sirve para dar una idea de la sintaxis de las mismas.

- **Instrucción de definición de variables**:

  Puede ser de dos tipos: Declaración o Inicialización. Permite crear una variable de tipo Int, Bool, Array o Struct asignándole un identificador. En particular, si es una inicialización, asigna un valor inicial a la variable en cuestión. Termina cuando se ponga el símbolo de punto y coma, al igual que en el lenguaje C++. Para las listas se deberá indicar su longitud (una constante numérica no negativa) en la declaración y también su tipo, que a su vez será Int, Bool, un tipo de Struct, o, recursivamente, otro Array. Esto último nos permite crear listas de varias dimensiones.

```
Int a = 5;
Bool b;
Array <Int,5> c = [1,2,3,4,5];
Array <Array <Int,5>,10> d; // d es una tabla de
                // enteros de 10 filas y 5 columnas
Array<MiStruct,3> a;
MiStruct m = {[1,2],False};
```

- **Instrucción de Asignación**:

  Permite asignar un valor de tipo Int, Bool, Array o Struct (devuelto por una expresión, una constante, una función u otra variable) a una variable del mismo tipo que haya sido previamente definida. Terminará con el símbolo de punto y coma.

```
x = 5;
y = (2 + 3) / 5;
z = x + y;
w = [1, 2, z];
```

- **Instrucción de Lectura:**

  Permite leer un valor del tipo Int, Bool, Array o Struct que indique el usuario, guardándolo en una variable. El valor leído se extraerá de línea de comandos. Para las listas y los Structs el orden de lectura será el usual, es decir, si declaramos una variable a de tipo Array < Array <Int, 2>, 2>, y queremos leer en a la lista [[1,2],[3,4]], debemos introducir por teclado la secuencia 1, 2, 3, 4. Y de igual manera, si queremos contruir el MiStruct $z = \{[1, 2], False\}$, se espera que se introduzca por teclado la secuencia 1, 2, 0.

```
read ( x );
```

- **Instrucción de Escritura:**

  Permite escribir un valor del tipo Int, Bool, Array o Struct en línea de comandos.

```
print ( True );
print ( x );
print ( [True, True] );
print ( funcion ( x ) );
```

- **Instrucción If:**

  Permite ejecutar un bloque de código distinto según el valor de unas ciertas condiciones booleanas. Distinguiremos tres tipos: una instrucción if sin cláusula else, otra instrucción if con cláusula else y una última instrucción if con múltiples posibles cláusulas elif y una posible cláusula else. Es importante destacar que exigiremos el uso de paréntesis para embeber la condición booleana y el uso de llaves para los bloques if, elif y else para evitar ambigüedades.

```
if ( True ) { x = 4; }
if ( x < 5 ) { x = x + 1; } else { x = x - 1; }
if ( x > 0 ) { } elif ( x < 0 ) { } else { }
```

- **Instrucción While:**

  Permite ejecutar un bloque de código de forma repetida mientras una cierta condición booleana es cierta.

```
while ( 2 + x == 3 ) { x = 2; }
while ( x > 1 ) { x = x + 1; y = x; }
```

- **Instrucción de Switch con Case**:

  Permite ejecutar un bloque de código distinto dependiendo del valor de una variable, que puede ser de tipo Int, Bool, Array o Struct. Es necesario introducir un bloque default (posiblemente vacío) al final de los cases, que es el que se ejecutará si la variable sobre la que se hace switch no es igual a ninguna de las expresiones. Las expresiones se comprueban en orden descendente y, una vez se hace cierta la igualdad con una expresión y se ejecuta su correspondiente bloque case, el switch termina. Es decir, se hace un salto automático fuera de la construcción switch.

```
switch ( x ) {
        case ( 0 ) {
                x = x + 1;
        }
        case ( 1 ) {
                x = x - 1;
        }
        default {
                x = 0;
        }
}
```

- **Instrucción de creación y definición de funciones**:

  Permite crear una función con parámetros de tipo Int, Bool, Array o Struct (pasados por referencia o por valor), un cierto código y un cierto valor de retorno, que también puede ser de tipo Int, Bool, Array o Struct. En caso de no haber valor de retorno o no haber parámetros de entrada, usaremos void. Además, tendremos una instrucción adicional de return para indicar el valor que se desea retornar. Dicho valor será el resultado de evaluar una expresión. Por defecto, los argumentos se pasarán por valor y para indicar el paso de una variable por referencia se utilizará el símbolo &.

```
def void fun ( Int & p1, Bool p2 ) { if ( p2 ) { print ( p1 ); } }
def Int aux ( void ) { fun (2, True); Int x = 2; return x - 1; }
```

4. Ejemplos Completos de Programas

Presentamos como última sección de este documento una serie de **ejemplos** completos de código de nuestro lenguaje. Están construidos según las reglas que se han descrito en anteriores secciones y utilizan todo el abanico de instrucciones, operadores y tipos propuestos. Todos ellos pueden probarse con nuestro compilador desde la carpeta *ejemplos*. Empezamos con un ejemplo sencillo de generación de números de Fibonacci que muestra definiciones de funciones, declaraciones, inicializaciones, instrucciones anidadas if y while, asignaciones, las instrucciones read y print, condiciones...

Listing 1: Generación de números de Fibonacci iterativo

```
1  def void main ( void ) {
2          Int aux = 1;
3          Int fib = 0;
4          Int lim;
5          Int init = 1;
6          read ( lim );
7          if ( lim > 0 ) {
8                  while ( init <= lim ) {
9                        print ( fib );
10                       aux = aux + fib;
11                       fib = aux - fib;
12                       init = init + 1;
13                 }
14         }
15         print(a);
16 }
```

En el ejemplo anterior el usuario introduce el número de términos de la sucesión de Fibonacci que desea generar. Como muestra el siguente programa, también podemos realizar el cálculo de manera recursiva. Este código ejemplifica el uso de variables globales, una llamada a función (incluida directamente en la instrucción print) y una instrucción elif:

Listing 2: Generación de números de Fibonacci recursivo

```
1  Int lim;
2  def void main ( void ) {
3          read ( lim ) ;
4          Int i=0;
5          while ( i <= lim ) {
6                  print ( fib(i) );
7                  i = i + 1;
8          }
9  }
10
11 def Int fib (Int n) {
12         if(n==0){
13                 return 0;
14         } elif(n==1){
15                 return 1;
16         } else{
17                 return fib(n-1)+fib(n-2);
18         }
19 }
```

A continuación, presentamos un ejemplo de ordenamiento de burbuja que va a servirnos para mostrar la creación y acceso a listas. Además, utilizaremos una función auxiliar que se encarga de intercambiar dos valores pasados por referencia.

Listing 3: Ordenamiento de Burbuja (Bubble Sort)

```
def Int main ( void ) {
        Int i = 1;
        Int n = 100;
        Bool sort = False;
        Array <Int,100> a;
        read ( a );
        while ( i < n && !sort) {
            i = i + 1;
            sort = True;
            Int j = 0;
            while ( j <= n-i ) {
                if ( a[j] > a[j+1] ) {
                    sort = False;
                    intercambiar(a[j],a[j+1]);
                }
                j = j + 1;
            }
        }
        print( a );
        return 0;
}
```

Listing 4: Intercambio de dos valores pasados por referencia a una función

```
def void intercambiar(Int & a, Int & b) {
        Int aux = a;
        a = b;
        b = aux;
}
```

El siguiente ejemplo es un contador de años bisiestos que muestra diversas operaciones booleanas y aritméticas, así como una llamada a función dentro de la condición de una instrucción if:

Listing 5: Contador de años bisiestos

```
def Int main ( void ) {
        Int count = 0;
        Int ini = 104;
        Int end = 643;
        while ( ini <= end ) {
            if ( bisiesto(ini) ) {
                count = count + 1;
            }
            ini = ini + 1;
        }
        print(count);
        return 0;
}
```

Listing 6: Identificador de años bisiestos

```
def Bool bisiesto (Int & a) {
        Bool dev;
        if ((a%4==0) && ((a%100!=0) || (a%400==0))) {
            dev = True;
        } elif ( False ) {
            print(dev);
        } else {
            dev = False;
        }
        return dev;
}
```

Seguimos con otro ejemplo de multiplicación de matrices, que muestra la creación de Structs con campos de tipo Array y el uso de los operadores . y []. Además, realiza instrucciones print sobre campos de tipo Array y trabaja con matrices, representadas como Arrays que tienen a su vez elementos de tipo Array.

Listing 7: Producto de matrices

```
Struct Matriz3x2{
        Array <Array <Int,2> ,3> t;
        Int filas;
        Int columnas;
};
Struct Matriz2x3{
        Array <Array <Int,3> ,2> t;
        Int filas;
        Int columnas;
};
Struct Matriz2x2{
        Array <Array <Int,2> ,2> t;
        Int filas;
        Int columnas;
};
Struct Matriz3x3{
        Array <Array <Int,3> ,3> t;
        Int filas;
        Int columnas;
};

def Int main (void){

        Matriz3x2 a;
        Matriz2x3 b;
        Matriz2x2 c;
        Matriz3x3 d;

        a.t=[[1,2+3],[2,3],[0,2]];
        a.filas=3;
        a.columnas=2;

        b.t=[[1,a.t[1][1],1],[4,0,1]];
        b.filas=2;
        b.columnas=3;

```

```
37          Int i=0;
38          while(i<a.filas){
39                  Int j=0;
40                  while(j<b.columnas){
41                          Int suma=0;
42                          Int k=0;
43                          while(k<a.columnas){
44                              suma=suma+a.t[i][k]*b.t[k][j];
45                              k=k+1;
46                          }
47                          d.t[i][j]=suma;
48                          j=j+1;
49                  }
50                  i=i+1;
51          }
52          print(d.t);
53          print(1111111111);   // Separador
54          Int i=0;
55          while(i<b.filas){
56                  Int j=0;
57                  while(j<a.columnas){
58                          Int suma=0;
59                          Int k=0;
60                          while(k<b.columnas){
61                              suma=suma+b.t[i][k]*a.t[k][j];
62                              k=k+1;
63                          }
64                          c.t[i][j]=suma;
65                          j=j+1;
66                  }
67                  i=i+1;
68          }
69          print(c.t);
70  }
```

El siguiente es un ejemplo que utiliza el operador $\wedge$ (la generación de código para este operador se ha realizado a mano, ya que no existe como operador predefinido de Web Assembly) de diversas manera, jugando con valores positivos y negativos.

Listing 8: Jugando con potencias

```
Int z = 3^(-2);

def Int main (void){
        print(z);
        print(pot(7));
        Int a =-2;
        Int b = 3;
        Int c = a^b;
        print(c);
        print(4^(-2));
        return 0;
}

def Int pot (Int n) {
       return n^2;
}
```

En relación a los dos ejemplos siguientes, el primero calcula la inversa de una lista introducida por el usuario que se pasa como argumento por referencia a una función, y el segundo utiliza una instrucción Switch con una expresión de tipo Struct.

Listing 9: Invertir una lista

```
def void main(void){
        Array<Int,10> l;
        read(l);
        Array<Int,10> delreves=l;
        reverse(delreves);
        print(delreves);
}

def void reverse(Array<Int,10>& lista){
        Array<Int,10> aux=lista;
        Int i=0;
        while(i<10){
                lista[i]=aux[9-i];
                i=i+1;
        }
}
```

Listing 10: Direcciones de Movimiento

```
Struct Dir {
        Int x;
        Int y;
};
Dir dirUser;
Array<Dir,4> direcciones = [{0,1},{0,-1},{1,0},{-1,0}];

def void main (void) {
        read(dirUser);
        switch(dirUser) {
                case(direcciones[0]) {
                        print(0);
                }
                case(direcciones[1]) {
                        print(1);
                }
                case(direcciones[2]) {
                        print(2);
                }
                case(direcciones[3]) {
                        print(3);
                }
                default {
                        print(-1);
                }
        }
}
```

Ahora incluimos otros dos ejemplos más, uno de uso algo más complejo de Structs y otro que resuelve el problema de las N reinas (N≤14) usando una gran variedad de operadores, instrucciones, paso de argumentos por referencia...

Listing 11: Jugando con Structs

```
Struct Camion {
        Int mercancia;
        Array<Int,2> productos;
};

Struct Flota {
        Array<Camion,3> camiones;
        Int gasolina;
        Bool operativa;
};

Flota f = {[{3,[24,25]},{4,[56,57]},{7,[31,32]}],100,True
};

def Int main(void) {
        // Imprime f original
        print(f);
        Array<Camion,3> nuevosCamiones =
[{9,[100,100]},{9,[200,200]},{9,[300,300]}];
        f.camiones = nuevosCamiones;
        f.operativa = !f.operativa;
        print(-1);
        // Imprime f nuevo
        print(f);
        print(-1);
        // Imprime diversos campos de f
        print (f.gasolina);
        print(f.camiones[2].productos[1]);
        // Imprime Struct tras llamada a funcion
        // con argumentos por referencia
        mercanciasFlota(f);
        print(f.gasolina);
}

def void mercanciasFlota(Flota & flota) {
        flota.gasolina = flota.gasolina - 50;
}
```

Listing 12: Problema de las N reinas

```
Struct Solucion{
        Array<Int,15> sol;
        Int contador;
};

Struct Columnas{
        Array<Bool,15> col;
        Int contador;
};

Struct Diagonales{
        Array<Bool,55> diag;
        Int contador;
};

def Int reinas(Solucion & sol, Int k, Columnas & col,
                Diagonales & diag, Int n){
        Int dev = 0;
        Int columna=1;
        while(columna<=n){
                sol.sol[k]=columna;
                if(!col.col[sol.sol[k]] &&
                !diag.diag[sol.sol[k]-k+n]
                && !diag.diag[k+sol.sol[k]+2*n-2]){
                   col.col[sol.sol[k]]=True;
                   diag.diag[sol.sol[k]-k+n]=True;
                   diag.diag[k+sol.sol[k]+2*n-2]=True;

                   if(k==n){
                   dev=dev+1;
                   }else{
                   dev=dev+reinas(sol,k+1,col,diag,n);
                   }

                   col.col[sol.sol[k]]=False;
                   diag.diag[sol.sol[k]-k+n]=False;
                   diag.diag[k+sol.sol[k]+2*n-2]=False;
                }
                columna=columna+1;
        }
        return dev;
}
```

```
43
44 def void main(void){
45         Int n;
46         read(n);
47         Solucion sol;
48         Columnas col;
49         Diagonales diag;
50         sol.contador=n;
51         col.contador=n;
52         diag.contador=4*n-2;
53
54         Int i=0;
55         while(i<=n){
56                 col.col[i]=False;
57                 diag.diag[i]=False;
58                 i=i+1;
59         }
60
61         while(i<=4*n-2){
62                 diag.diag[i]=False;
63                 i=i+1;
64         }
65
66         i=0;
67         while(i<=n){
68                 print(reinas(sol,1,col,diag,i));
69                 i=i+1;
70         }
71 }
```

Presentamos un ejemplo más complejo que muestra la potencia de nuestro lenguaje de programación en cuanto a paso de argumentos de tipo Array o Struct a funciones, expresiones compuestas de múltiples operadores y tipos de datos, funciones con valores de retorno de cualquier tipo (Int, Bool, Array o Struct)...

Listing 13: Ejemplo de paso de argumentos, return y llamadas a funciones

```
1 Struct S {
2         Int x;
3         Array<Int,3> j;
4 };
```

```
5
6  Int z = 44;
7
8  S s = {233, [12,13,14]}; // Inicializacion de Struct
9
10 def void main (void){
11         // Inicializacion de Array con llamada a funcion
12         Array<Int,4> b = func2();
13         S prueba;
14         // Acceso a campos del Struct
15         prueba.x = 2;
16         prueba.j = [1,2,3];
17         // Llamada a funcion como argumento de otra
18         S m = func1(True, b, func3(z), prueba);
19         print(b);
20         print(m.x);
21         print(m.j);     // print de un Array
22 }
23
24 // Argumentos de diversos tipos
25 def S func1 (Bool b, Array<Int,4> a, Int m, S prueba) {
26         print(a);
27         print(prueba.j);
28         if (b) { print(m); } else { }
29         print(z);       // Uso de variable global
30         return s;
31 }
32
33 // Valor de retorno de tipo Array
34 def Array<Int,4> func2(void) {
35         Array<Int,4> a = [40,39,21,45];
36         return a;
37 }
38
39 def Int func3 (Int n) {
40         return n + 2;
41 }
```

Por último, hemos incluido varios ejemplos adicionales en la carpeta *ejemplos*, que dejan ver cómo nuestro compilador maneja errores sintácticos, errores de vinculación y errores de tipado.

Queremos destacar dos cosas antes de concluir este documento. En primer lugar, **el compilador que entregamos es, hasta donde hemos probado, plenamente funcional**. Todos los ejemplos que se presentan en este documento han sido probados y al ejecutarlos hacían lo que tenían que hacer. Además, en todas las pruebas que hemos hecho desde que finalizamos el desarrollo del mismo no hemos encontrado errores. En segundo lugar, nos gustaría poner en valor que **hemos hecho todo lo que nos comprometimos a hacer en nuestro documento inicial e incluso hemos introducido algunas cosas extra** que no aparecían en dicho documento como los Structs. Podemos decir que estamos contentos con el resultado que entregamos.