Práctica de PROCESADORES DE LENGUAJES

LENGUAJE CC

Carlos Morán Alfonso cmoran01@ucm.es

Carlos Tardón Rubio ctardon@ucm.es

Universidad Complutense de Madrid

Índice

1. Introducción al lenguaje

Partimos de C básico con algún añadido.
Usaremos **;** para separar lineas distintas (para eliminar posibles ambigüedades futuras, aunque todavía no sabemos si será necesario), y permitimos comentarios simples (una sola línea) con //. Ejemplo:

```
Linea1; // Esto es un comentario
Linea2; // Cada linea acaba en ;
```

Posiblemente la llave final de los bloques (tanto anidados como bloques de bucles, funciones, etc) no necesite el ***;***

En azul los cambios

2. Requisitos

2.1. Identificadores y Ámbitos de Definición

En cuanto a los identificadores, el compilador permite llamar a dos cosas por el mismo nombre, prevaleciendo la que aparezca en último lugar (en el caso de las funciones, esto significa que no hay sobrecarga, sino que se aplica la última definición de la función. En el caso de declaraciones de variables, el identificador se vincula con la última declaración que aparezca antes del uso del identificador). En este caso, el compilador emitirá un **warning** pero seguirá la compilación, dejando a elección del usuario si cambiar los nombres o atenerse a las posibles consecuencias.

Mínimos: Declaración de variables simples y de arrays de cualquier tipo, incluidos otros arrays (o bien, permitir arrays de varias dimensiones). Bloques anidados (que requerirá trabajar con tabla de símbolos para bloques anidados).

Tanto para **tipos simples como para arrays de cualquier tipo**, usaremos dos sintaxis:

```
tipo nombreVariable;                // declaracion sin valor inicial
tipo nombreVariable = valorInicial; //declaracion con valor inicial
```

Ejemplo para variables simples de tipo int:

```
int a;                // declaracion sin valor inicial
int a = 0;            //declaracion con valor inicial
```

Sintaxis de arrays (hemos cambiado un poco la sintaxis original de C):

```
tipo[N] nombreArray;     // array simple. N tiene que ser un entero
    constante

tipo[N] nombreArray = {N posiciones};   //declaracion con valor inicial

int[4] a;               // Ejemplo array simple sin valor inicial
int[4] a = {1,2,3,4}; // declaracion con valor inicial.
                      // La lista de la derecha tiene que tener 4 posiciones
```

Sintaxis de arrays multidimensionales (Cambia un poco la sintaxis original de C):
Nota: en arrays estáticos, N, M, K tienen que ser constantes enteras.

```
tipo[N][M] nombreMat;     // Array doble (array cuyos elementos son arrays)
tipo[N][M][K] nombre     //Array multidimensional

int[4][5] a;               // Ejemplo sin valor inicial
int[2][3][4] b;
```

¿Que pasa si tenemos matriz int[5][5] mat y accedemos a mat[i]? Devuelve un array de tamaño 5
Asumimos que puede haber arrays de cualquier numero de dimensiones, int[N][M][K][H]...
Los arrays sin valor inicial están inicializados al valor inicial de sus componentes.
Nota: En las funciones, los arrays se pasan por valor(copiando todo su contenido), por lo que en el chequeo de tipos comprobará si el array que pasamos tiene los mismos tamaños que el array esperado.

Las declaraciones de arrays con valor inicial(es decir, esto: int[3] v = {1,2,3}), hemos generado el lexico, la sintaxis, el ast, la vinculacion y el chequeo pero no generamos su código.

Bloques anidados, demarcados mediante llaves:

```
{
    {
        {
        .
        .
        .
        }
    }

}
```

Asumimos que puede haber bloques anidados de cualquier profundidad.

Opcionales: Punteros, registros, funciones, clases (sin ningún tipo de herencia), módu-

los, cláusulas de importación.

Punteros: Con la sintaxis de C habitual. Un * a la derecha del tipo indica tipo puntero. Operadores unarios & y * para referenciar y dereferenciar. Al igual que en C, el compilador no se preocupa de que las referencias sean accesibles, así que tiene que ser el programador el que no devuelva referencias a variables locales:

```
tipo* puntero1; // puntero sin asignacion inicial
tipo * puntero2 = &variableDelTipo; //puntero con asignacion inicial y
    operador referencia.
tipo * puntero3 = new tipo;

int a = 3;
int* punt = &a;
3 + *punt;                // Operador dereferenciar. Tiene que dar 6
int** punToPun = &punt; //Puntero doble.

int[3] v = {1,2,3};
int[3] *pv = &v;          // puntero a array
int cuatro = (*pv)[1] + 2; //ejemplo dereferenciar puntero a array
```

Puede haber punteros dobles, triples, etc.

Funciones: con la sintaxis habitual. void si no retorna nada. Es un tipo,pero no podremos declarar variables del mismo, solo vale como retorno

```
tipoRetorno nombreFuncion(tipo1 arg1, , tipon argn){ //Bloque local, con
    tabla de simbolos
    return variableRetorno; // Palabra reservada return para retornar un
    valor
}
```

Asumimos paso por valor. Todo se pasa por valor.

Clases y structs: con la sintaxis de class Java, por lo que un struct es simplemente un class sin métodos, y con la sintaxis de C++ para creacion de instancias (no hace falta new pues no se reserva memoria dinamica). Las clases son tipos nuevos. Una clase tiene que tener como mínimo un atributo, y puede tener 0 o más funciones. Primero van las declaraciones de atributos y luego las funciones, no permitiendose intercalar ambas.

```
class NombreStruct{ //Struct, que es una clase sin metodos
    int a;      //Atributo del struct
    int b ;

} // Bloque

class NombreClase{s
    int a;
    void funcion(int b){
        a      //Opcion 1 para acceder al valor de a, que es como accederiamos con bloques anidados
        this.a // Sintaxis opcional para acceder al valor de a. No sabemos si incluir esta opcion
    }

}

NombreClase obj1; //Creacion de un objeto. No hace falta new para instanciarlo
obj1.a          //para acceder al atributo a
obj1.funcion(2)  //para acceder al metodo funcion del objeto
```

Los atributos de las clases no pueden tener valor por defecto

Modulos: Se pueden incluir los modulos que se quieran, pero con la restriccion de que el grafo de importacion sea un arbol (no hay ciclos). Además los includes solo se permiten en las primeras líneas (de momento).

```
#include "nombreArchivo.cc";
```

Si se importa varias veces un fichero, este solo se carga una vez y se añade una vez. Esta mejora se debe a la añadidura de un set que actúa como caché de modulos ya cargados. Si se importa un archivo varias veces, el compilador emite un warning.

Al importar un módulo, se inserta el código del módulo importado cuando aparece la instrucción import. De esta manera, los módulos no son ámbitos nuevos, pertenecen al ámbito del módulo principal. Permitimos código en los módulos importados, pues pensabamos ejecutarl en el orden de importación. En el siguiente ejemplo, primero se ejecutaría el código del modulo1, luego el del modulo2, y luego el código del archivo principal. Sin embargo, el código que generamos solo considera modulos de definiciones

Añadimos la necesidad de que haya un main en el módulo principal. El código fuera de las funciones se chequeará, pero no se generará su código wasm

```
#include "modulo1.cc";
#include "modulo2.cc";

```

```
4 //Codigo del archivo principal
```

NamingConvention:

- Variables y punteros: mayúsculas, minúsculas y números, empezando por letra minúscula
- Clases: mayúsculas, minúsculas y números, empezando por letra Mayúscula
- Funciones: mayúsculas, minúsculas y números, empezando por letra minúscula seguidos de un abre paréntesis, una serie de argumenos y un cerrar paréntesis
- Modulos: camelCase, solo letras mayusculas y minusculas, con extension .CC (Carlos y Carlos)

2.2. Tipos

Mínimos: Declaración explícita del tipo de las variables. Tipos básicos predefinidos enteros y booleanos. Operadores infijos, con distintas prioridades y asociatividades para estos tipos. Tipo array. Equivalencia estructural de tipos. Comprobación de tipos.

Tipos predefinidos: entero, booleano, float y caracter, que se pueden representar de la siguiente forma:

```
1 int a = 5;   // variable entera con id a
2 bool b = true; // variable booleana con id b
3 char c = '5'; // caracter con id c
4 float d = 5.0; // numero real con id b
```

Si en la declaración de la variable no aparece un valor inicial, estarán inicializadas al valor por defecto(0 para ints, false para bool, '\u0000' para char y 0.0f para float)

El léxico, la sintaxis, el ast, vinculacion y el chequeo de tipos está creado para los chars, y los floats, pero no implementaremos la generación de código para chars ni floats.

Operadores unarios predefinidos: números negativos, negación de booleanos, referencia y dereferencia(para punteros)

```
1 !true;              // false
2 -3;                 // -3
3 &a                   // direccion de memoria de a
4 *punt;               // Operador derreferenciar. punt tiene que ser un
      identificador de variable
5
```

```
!condicion[i]      //Expresion formada por varios operadores con distintas
    asociatividades
```

Operadores infijos predefinidos: Según tipos:

- Enteros: Suma, resta, multiplicación, división, potencia, módulo, incremento y decremento en 1.
- Reales: Las mismas que los enteros menos el módulo.
- Booleanos: AND, OR y NOT lógicos
- Chars: de momento ninguna, aunque valoraremos si incluir algún tipo de append (entre chars y/o array de chars) (por ejemplo con el operador #, para que 'a' # 'b' # 'c' devuelva un array de chars).
- Arrays: Acceso a un subíndice de un array
- Todos: Operador referenciar, y dereferenciar, comparaciones (>,>=,<,<=,==,! =) entre variables del mismo tipo, paréntesis de prioridad, uso de ':' para cambiar la asociatividad de otro operador

```
3+5;    // 8
5-3;    // 2
5*3;    // 15
5/3;    // 1 o 2 (dependiendo de rendodeo)
10.0/4  // 2,5
5+4/3; // 6 o 7 (dependiendo de redondeo)
(5+4)/3; // 3
3^3 // 27
5%3 // 2

true && true  // true
true || false // true
true == false // false
true != false // true
7 == 7        // true
7 != 7        // false
7 <= 9        // true
7 < 5         // false
7 >= 9        // false
7 > 5         // true
```

Tabla de prioridad de operadores (0 es la menor prioridad y 12 la mayor).

Operador	Descripción	Tipo	Asociatividad	Prioridad
||	Or lógico	Binario infijo	Asociativo a izquierda	0
&&	And lógico	Binario infijo	Asociativo a izquierda	1
==	Igualdad lógica	Binario infijo	Asociativo a izquierda	2
!=	No igualdad lógico	Binario infijo	Asociativo a izquierda	2
<, <=	Desigualdad lógica	Binario infijo	Asociativo a izquierda	3
>, >=	Desigualdad lógica	Binario infijo	Asociativo a izquierda	3
+	Suma aritmética	Binario infijo	Asociativo a izquierda	4
-	Resta aritmética	Binario infijo	Izquierda a derecha	4
-:	Resta aritmética 2.0	Binario infijo	Derecha a izquierda	5
*	Producto aritmético	Binario infijo	Asociativo a izquierda	6
/	División aritmética	Binario infijo	Izquierda a derecha	6
%	Módulo	Binario infijo	Izquierda a derecha	6
/:	División aritmética 2.0	Binario infijo	Derecha a izquierda	7
%:	Módulo 2.0	Binario infijo	Derecha a izquierda	7
^	Exponente	Binario infijo	Izquierda a derecha	8
^:	Exponente 2.0	Binario infijo	Derecha a izquierda	9
++, - -	Incrementar/decrementar uno	Unario prefijo	No asociativo	10
-	Opuesto de un entero	Unario prefijo	No asociativo	10
*	Operador dereferenciar	Unario prefijo	Asociativo a derecha	10
&	Dirección de memoria apuntada	Unario prefijo	No asociativo	10
!	Not lógico	Unario prefijo	Derecha a izquierda	10
++, - -	Incrementar/decrementar uno	Unario postfijo	No asociativo	11
[]	Subíndice en un array	Unario postfijo	Asociativo	11
()	Llamada a funcion	Unario postfijo	Asociativo	11
.	Acceso a miembro de instancia	Binario infijo	Asociativo a izquierda	11

Opcionales: Tipos con nombre y definición de tipos de usuario.

Introducimos también los tipos enumerados y la palabra clave using. Sintaxis:

```
enum tipoEnum = {valor1,valor2,...,valorn};
using li = int[3]; //funciona como el typedef, es decir, es el mismo tipo
    que int[3]
```

Los enumerados se tratan internamente como enteros consecutivos, empezando en 0. Hemos generado el lexico, la sintaxis, el ast, la vinculacion y el chequeo de tipos de los enums, pero no generamos su código. Los using sí que están implementados totalmente.

Ejemplo de uso:

```
enum direcciones = {ARRIBA,ABAJO,IZQUIERDA,DERECHA}; // Un tipo enumerado
    llamado direcciones
direcciones d = ARRIBA; // creamos una variable del tipo enumerado
    direcciones y le damos el valor ARRIBA

using ll = int[4];
ll n = 30; // Creamos una variable n con nuestro nuevo alias
```

2.3. Conjunto de Instrucciones del Lenguaje

Mínimos: Instrucción de asignación incluyendo elementos de arrays, condicional con una y dos ramas, y algún tipo de bucle. Expresiones formadas por constantes, identificadores con y sin subíndices (para acceso a arrays) y operadores infijos.

Instrucciones predefinidas:

- Instrucción de asignación " = ": Tanto para asigar o cambiar el valor de una variable. Junto con el operador de subíndice, también vale para cambiar el valor de un elemento del array

```
int a;
a = 5;
int[2] v; // declaracion de array. Todos los arrays tienen valor por
    defecto
v[0] = 3; // asignacion de valor a un elemento del array v.
```

- Sentencias condicionales: De una rama (if) o dos (if-else).

  En los if, else, while y for, tanto el bloque de instrucciones (si va con llaves), o la instrucción inmediatamente de después (si es sin llaves), pertenecen a un ámbito nuevo

```
if(expresionBooleana){ // condicional de una rama, con sintaxis de
    llaves
    sentencia1;
    sentencia2;
    ...
}

if(expresionBooleana){ // condicional de dos ramas, con sintaxis de
    llaves
    sentencia1;
    sentencia2;
    ...
}else{
```

```
    sentencia3;
    sentencia4;
    ...
}
```

```
if(expresionBooleana) // condicional de una rama, con sintaxis sin
    llaves. Solo ejecuta la siguiente linea
    sentencia1;


if(expresionBooleana) // condicional de dos ramas, con sintaxis sin
    llaves
    sentencia1;
else
    sentencia2;
```

- Bucles: for y while con la misma sintaxis de C.

```
while(expresionBooleana){ //bucle while
    sentencia1;
    ...
}

for(asignacionInicial;expresionBooleana;asignacionFinal){ // for
    sentencia1;
    ...
}
```

En los bucles for, la declaración inicial pertenece al ámbito de arriba. De esta forma, la variable ahí declarada es visible después de que termine el cuerpo del for

```
while(expresionBooleana) //bucle while sin llaves
    sentencia1;

for(asignacionInicial;expresionBooleana;asignacionFinal) // for sin
    llaves
    sentencia1;
```

- Operador () para llamar a una función usando la convención ya establecida
- Instrucción return, que devuelve un valor de una función. Es obligatoria, pero no hace falta ponerla al final.

```
tipoFun nombreFun(arg1,arg2,...,argn){
    tipoFun a;
    ...
    return a;
}

tipoFun c = nombreFun(a1,a2,...,an);
```

- Interacción con el usuario: print (para output) y prunt (para input). La sintaxis es, de momento)

```
print("Texto sin comillas")  //El texto de print se tratara
    internamente como un array de chars
prunt(x);                    // Donde x es un identificador de variable
```

- Expresiones con punto y coma. Solo permitimos el uso de ++, –, (exclusivamente sobre identificadores de variables) y llamadas a funciones(sobre identificadores de función, funciones miembro de objetos, etc)

```
++i; // Expresion que incrementa el valor de i
i--; // Expresion que decrementa el valor de i

f(); // Llamada a funcion sin argumentos
obj.a[3].f(); // Llamada a funcion (miembro de un objeto dentro de un
    array de objetos) que es atributo de obj.
```

- Instrucciones break y continue

```
while(expresionBooleana){
    ...
    if(condicion){
        continue; // instruccion de salto para saltarse el resto de
    esta iteracion del bucle
    }
    ...
}

while(expresionBooleana){
    ...
    if(condicion){
        break; //instruccion de salto para salir del bucle
    }
    ...
}
```

Expresiones:

```
// Ejemplos de expresiones
3       // expresion constante
a        //expresion simple formada por un identificador de variable
v[i]    //expresion simple formada por identificador de array con subindice
3 + 3   //expresion compuesta de constantes y un operador infijo
a * 5   //expresion compuesta de constante, identificador de variable, y un
    operador infijo
v[i] * 5 + a //expresion mixta
```

Ejemplo de asignaciones, condicionales, bucles y return:

```
// Programa de ejemplo factorial.cc

int factFor(int n){
    int fact; // Inicializamos fact
    fact = 1; // le asignamos el valor 1
    for(int i = 2; i <= n; ++i)
        fact = fact * i;

    // La sintaxis de for es como la de C++ con for(variable que se va a
    modificar, condiciones, modificacion de la variable)
    return fact; // Devolvemos fact con la instruccion return
}

int factWhile(int n){
    int fact; // Inicializamos fact
    fact = 1; // le asignamos el valor 1
    int i = 2; // creamos la variable
    while(i <= n){
        fact = fact * i;
        i++;
    }
    // La sintaxis de while es como la de C++ con while(condiciones)
    return fact; // Devolvemos fact con la instruccion return
}

int factRecursivo(int n){
    if(n == 1)
        return 1;
    else{
        return n*factRecursivo(n-1);
    }
}

int main(){
    int n = 3;

    int a = factFor(n);
    int b = factWhile(n);
    int c = factRecursivo(n);
    // a = b = c = 6


    return 0;
}
```

Opcionales: Expresiones con punteros y nombres cualificados (en presencia de clases o registros). Instrucción case o similar con salto a cada rama en tiempo constante. Llamadas a funciones, o/y métodos de clase. Instrucciones de reserva de memoria dinámica.

```
// Ejemplos de expresiones
*p                                  // expresion con puntero (dereferenciar)
nombreObjeto.nombreAtributo         //expresion con nombre cualificado
*p + nombreObjeto.nombreAtributo     //expresion compuesta


funcion(2,3); //llamada a la funcion funcion

NombreClase obj1; //Creacion de un objeto. No hace falta new para instanciarlo
obj1.a          //para acceder al atributo a
obj1.funcion(2)  //para llamar al metodo funcion del objeto

//Ejemplo reserva memoria dinamica con new
int[] v = new int[size];          // donde size es una constante o identificador de variable entera
                                  // nota: en arrays estaticos no permitimos la sintaxis int[]
```

la reserva dinamica de memoria está implementada la sintaxis, semántica y chequeo de tipos, pero no generamos código wasm

Switch-case con sintaxis un poco distinta a la habitual de C:

```
switch(identificadorVariableEnumerable) {
    case(1){
        // code block
    }
    case(2){
        // code block
    }
    default{
        // code block
    }
}
```

El switch no es en tiempo constante, pero nos hemos asegurado de calcular una sola vez la condicion

2.4. Gestión de Errores

Mínimos: Indicación del tipo de error, fila y columna. Parar la compilación. Recuperación de errores (tratar de proseguir la compilación tras un error, a fin de detectar más errores).

En la última versión hemos añadido un código de errores: un asterisco (seguido de un mensaje de error) significa error léxico. Dos asteriscos error sintáctico. Tres asteriscos error semántico.

2.4.1. Ejemplo de errores léxicos

```
void main(){
    int[3] a_; // caracter extra _
    int[3] b =};
    int[3] c;

    int i_ = 0; // caracter extra _
    while(i < 3){
        c[i] = a[i]*b[i];
        i++;
    }
}
```

2.4.2. Ejemplo de errores de vinculación

```
void main(){
    int a = 7;
    int b = 9;
    int b = a; // Warning semantico por varios identificadores 'b'. Nos
    quedamos con el ultimo
    int c = d; // Error semantico por identificador d no encontrado
    ll e = 17; // Error semantico por identificador de tipo no encontrado
    ll
}
```

2.4.3. Ejemplo de errores de chequeo de tipos

```
class Struct{
    char a;
    bool b;
    int c;
}

class Clase{
    int d;
    bool e;
    char f;

    void fun1(int a, int b){
        print(a);
    }

    int fun2(){
        return 3;
    }

```

```
20      int fun3(int v, int w){
21          return v+w;
22      }
23
24      void fun4(){
25          return 9; // Error semantico. Funcion void con return
26      }
27
28      int fun5(){
29          int a = 7;
30          if(a < 5){
31              return 9;
32          }
33          return 'l'; // Error semantico: tipos de funcion y de retorno
        distintos
34      }
35  }
36
37
38  void main(){
39      int a = 7;
40      int b = 9;
41      int *punt = a+b; // Error semantico: Tipos disintos a ambos lados de la
         declaracion (int y puntero a int)
42      Struct cl1;
43      Clase cl2;
44      bool c = 'l'; // Error semantico: Tipos distintos a ambos lados de la
        igualdad de la declaracion (bool y char)
45      bool d = true;
46      cl1.a = false; // Error semantico. Tipos distintos a ambos lados de la
        igualdad (Struct.a es char y no bool)
47
48      cl2.fun1(); // Error semantico: Numero de argumentos incorrecto para
        Clase.fun1()
49      int e = cl1.fun1(a,b); // Error semantico: Clase.fun1 no es int
50  }
```

Cada error nos tiene que dar la fila y la columna, tanto en errores léxicos como sintácticos o semánticos.

3. Programas de Ejemplo

Nota: En varios de los ejemplos hacemos print(11111) o print(-1) como una especie de 'línea en blanco' ya que no generamos código para elementos que no sean enteros y print introduce un salto de línea directamente

3.1. Ejemplo 1: Recursión

```
void recursiva(int a){
    print(a);
    if(a <= 4)
        recursiva(a + 1);
    print(a);
}
void printea(int b){
    print(b);
}
int dameUnDos(int a){
    return 2;
}


void main(){
    recursiva(1);
    print(111111111); // para separar
    print(dameUnDos(dameUnDos(3)));
    print(111111111); // para separar
    recursiva(dameUnDos(222222));
}
```

3.2. Ejemplo 2: Include, operaciones aritméticas, using, bucles y switch

Generamos código solamente para +,-,*,/ y las operaciones 2.0 -: y /: (ni módulo ni exponente)

pruebaInclude.cc:

```
void includedVoidFunction(){
    print(99);
}

int includedIntFunction(int v, int w){
    return v*w*(v+w);
}
```

```
#include "test/pruebaInclude.cc";

using ent = int;



void main(){
    ent n = 49;
    print(n);
    int a = 18;
    int b = 180;
    tokFunction();
    for(int i = 0; i < 6; ++i){
        switch(i){
            case(0){
                print(a-b);
            }
            case(1){
                print(a-:b);
            }
            case(2){
                print(a/b);
            }
            case(4){
                print(a/:b);
            }
            case(3){
                continue;
            }
            default{
                break;
            }
        }
        print(i);
    }
    includedVoidFunction();
    int sol = includedIntFunction(a,b);
    print(sol);
}
```

3.3. Ejemplo 3: Bucles, recursión y declaración y llamada a funciones

```
int factFor(int n){
    int fact;
    fact = 1; // le asignamos el valor 1
    for(int i = 2; i <= n; ++i)
        fact = fact * i;

    return fact;
}

int factWhile(int n){ // puntero como parÃ¡metro
    int fact;
    fact = 1;
    int i = 2;
    while(i <= n){
        fact = fact * i;
        i++;
    }
    return fact;
}

int factRecursivo(int n){
    if(n == 1)
        return 1;
    else{
        return n*factRecursivo(n-1);
    }
}

void main(){
    int n = 3;

    int a = factFor(n);
    int b = factWhile(n);
    int c = factRecursivo(n);
    int d = factRecursivo(factRecursivo(n));
    print(a);
    print(b);
    print(c);
    print(d);
}
```

3.4. Ejemplo 4: Bucles, arrays (creación y acceso), matrices, punteros y Clase

```
1
2  int[3][3]* mulMat(int[3][3] a, int[3][3] b){
3      int[3][3] c;
4      for(int i = 0;i < 3; ++i){ //version con for. i es variable local
       bloque funcion
5          for(int j = 0;j < 3;++j){//j variable local bloque anidado 1
6              c[i][j] = 0;
7              for(int k = 0;k<3;++k) {// k variable local bloque anidado 2
8                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
9              }
10         }
11     }
12     int[3][3]* punt = &c;
13     return punt;
14 }
15 // Producto de matrices. Contenidos: clase con atributos y metodo, acceso a
       atributos desde el metodo,bucles anidados, y acceso a puntero desde
       funcion
16 class MatMult{
17     int[3][3] a;
18     int[3][3] b; //atributos.
19
20     void multMat(int[3][3]* c){
21         for(int i = 0;i < 3; ++i){
22             for(int j = 0;j < 3;++j){
23                 (*c)[i][j] = 0;
24                 for(int k = 0;k<3;++k) {
25                     (*c)[i][j] = (*c)[i][j] + a[i][k] * b[k][j]; //
       accedemos a atributos desde el metodo
26                 }
27             }
28         }
29     }
30 }
31
32 void printMatriz(int[3][3] mat){
33     for(int k = 0; k < 3; ++k){
34         for(int j = 0; j < 3; ++j){
35             print(mat[k][j]);
36         }
37         print(-1);//para separar
38     }
39 }
40
41
42 void main(){
43     int[3][3] a;
44     int[3][3] b;
45     for(int i = 0; i < 3; ++i){
46         for(int j = 0; j < 3; ++j){
47             a[i][j] = 3*i+j+1;
48             b[i][j] = 3*(3-i)-j;
49         }
```

```
50      }
51      printMatriz(a);
52      printMatriz(b);
53      int[3][3]* prod = mulMat(a,b);
54      // prod = {{1,2,3},{4,5,6},{7,8,9}}*{{9,8,7},{6,5,4},{3,2,1}} =
        {{30,24,18},{84,69,54},{138,114,90}}
55      printMatriz(*prod);
56      print(111111); // para separar
57
58      //hacemos lo mismo, pero usando la clase
59      int[3][3] result;
60      MatMult m; //Declaracion y creacion instancia de clase MatMult
61      for(int k = 0; k < 3; ++k){
62          for(int j = 0; j < 3; ++j){
63              m.a[k][j] = 3*k+j+1; //establecemos el valor inicial de los
        atributos del objeto
64              m.b[k][j] = 3*(3-k)-j;
65          }
66      }
67      printMatriz(m.a);
68      printMatriz(m.b);
69      m.multMat(&result);  //acceso a metodo mulMat del objeto m, pasando
        referencia de result
70      printMatriz(result);
71  }
```

3.5. Ejemplo 5: Clases, llamada a métodos, recursión en métodos y arrays

```
1
2  // Ejemplo de Fibonacci. Contenidos: acceso a arrays y punteros, pasar
       punteros como parÃ¡metos, acceso a objetos
3
4  class Fibonacci{
5      int[200] arrSol;
6
7      int nesimo(int n){ // devuelve el n-esimo numero de Fibonacci
8          if((n == 0) || (n == 1))
9              return 1;
10         int sol = nesimo(n-1) + nesimo(n-2);
11         return sol;
12     }
13
14     void nprimeros(){ // Funcion que pone en arrSol los 200 primeros
       numeros de Fibonacci
15         for(int k = 0; k < 200; ++k){
16             if((k == 0) || (k == 1)){
17                 arrSol[k] = 1;
18             }
19             else{
20                 int num = arrSol[k-1] + arrSol[k-2];
21                 arrSol[k] = num;
22             }
23         }
24     }
25 }
26
27
28 void main(){
29     Fibonacci fib;
30     int n1 = 20;
31     int n2 = 10;
32     int n3 = 30;
33
34     int sol = fib.nesimo(n1);
35     print(sol);
36     print(-1); // Usamos -1 como salto de lÃnea improvisado
37     fib.nprimeros();
38     for(int i = 0; i <= n1; ++i)
39         print(fib.arrSol[i]);
40     print(-1);
41     sol = fib.nesimo(n2);
42     print(sol);
43     print(-1);
44     for(int j = 0; j <= n2; ++j)
45         print(fib.arrSol[j]);
46     print(-1);
47     sol = fib.nesimo(n3);
48     print(sol);
```

```
49      print(-1);
50      for(int k = 0; k <= n3; ++k)
51          print(fib.arrSol[k]);
52  }
```

3.6. Ejemplo 6: Todo junto

```
class NombreStruct{
    int b;
    int a;
    int c;
    int lechuga(){
        a = 16;
        print(1);
        return 1;
    }
}

class ClaseRara{
    int y;
    NombreStruct hello;
    int u;
    int lechuga(){
        y = 99;
        print(9);
        return 9;
    }
    void tomate(ClaseRara cr2){
        cr2.u = 1;
        u = 66;
    }
}
int lechuga(){
    print(0);
    return 0;
}


int funcion( int[2] v, int[2][2] mat,ClaseRara cr){
    for(int i = 0;i < 2;++i)
        print(v[i]);
    for(int i = 0;i < 2;++i){
        for(int j = 0;j < 2;++j){
          print(mat[i][j]);
        }
    }
    print(cr.hello.a);
    print(cr.hello.b);
    print(cr.hello.c);
    print(cr.y);
    return 0;
}

void main(int c){
    int[2][2] mat;
    for(int i = 0;i < 2;++i){
        for(int j = 0;j < 2;++j){
          mat[i][j] = 2*i + j + 10;
```

```
52          }
53      }
54      NombreStruct he;
55      he.a = 4444;
56      he.b = 6666;
57      he.c = 2000;
58      ClaseRara cr;
59      ClaseRara cr2;
60      cr.hello.a = 4545;
61      cr.hello.b = 3434;
62      cr.hello.c = 1212;
63      cr.y = 454545;
64      cr2.y = 0; cr2.u = 0;
65      cr2.hello.a = 0;
66      funcion(mat[0],mat,cr);
67      print(1111111111);
68      print(lechuga());
69      print(cr.lechuga());
70      print(cr.hello.lechuga());
71      print(111111);
72      print(cr.y);
73      print(cr.hello.a);
74      print(cr2.y);
75      print(cr2.hello.a);
76      cr.tomate(cr2);
77      print(cr.u);
78      print(cr2.u);
79
80  }
```