Entrega 1: Práctica Procesadores de Lenguajes

- **Bittor Alaña Olivares**
- **Francisco Javier Blázquez Martínez**

1. ESTRUCTURA GENERAL DE UN PROGRAMA

En primer lugar, aclaramos en qué consiste un programa. Un programa en nuestro lenguaje es una serie de definiciones de funciones y procedimientos con una función especial de tipo int, que debe ser la última declarada y debe tener por indentificador la palabra **main**. La ejecución del programa consiste en ejecutar una por una y en el orden en el que se presentan todas las instrucciones de la función main.

La declaración de procedimientos y funciones obliga a especificar los parámetros y el comportamiento de conjuntos de instrucciones y nombrar estos con un identificador para poder ser invocados posteriormente. Los procedimientos se declaran con la palabra reservada **void** y las funciones tienen que definir explícitamente su tipo en su declaración. La sintaxis de estas definiciones es la siguiente:

Declaración procedimiento	**void identificador(tipo id, ... , tipo id) { instrucciones }**
Declaración función	**tipo identificador(tipo id, ... , tipo id) { instrucciones return exp;}**

Donde **identificador** e **id** hacen referencia a cadenas de caracteres alfanuméricos o '_' consecutivos, **tipo** hace referencia a un tipo permitido por el lenguaje (ver punto 2), **exp** hace referencia a una expresión del mismo tipo que el de la declaración de la función (ver punto 3) y **instrucciones** hace referencia a una lista de instrucciones válidas del lenguaje (ver punto 4).

Otras características de nuestro lenguaje es que es un lenguaje *case sensitive*, que considera a todos los efectos iguales espacios, saltos de línea y tabuladores (empleados como separadores) y que permite comentarios de línea al estilo C, indicados con dos barras consecutivas '//'.

2. TIPOS

Nuestro lenguaje contempla constantes, variables y retornos de funciones de cualquiera de los siguientes tipos:

- **Entrero:** Se identifican con la palabra reservada **int.**
- **Booleano:** Se identifican con la palabra reservada **bool**.
- **Punteros:** Se pueden declarar punteros de cualquier tipo. Se identifican con el asterisco ***** precediendo a un tipo válido cualquiera.
- **Registros:** Consisten en agregaciones de tipos en tuplas. Se identifican con una lista ordenada de tipos separadas por comas entre los símbolos '<' y '>' al principio y final respectivamente. El acceso a un elemento en concreto de una variable de tipo registro se hace indicando el índice del elemento (en la declaración del registro, empezando a contar por cero) entre estos mismos símbolos '<' y '>'.
- **Arrays:** Consisten en elementos de un mismo tipo consecutivos en memoria. En su declaración exigimos que el número de elementos sea explícito, esto es, no soportamos directamente arrays de dimensión variable. Se identifican con un tipo válido (posiblemente

un tipo array), y un tamaño válido (entero mayor o igual que uno), en ese orden, dados entre corchetes cuadrados de apertura y cierre y separados por coma.

La sintáxis de tipos en nuestro lenguaje es por tanto la siguiente (**tipo** es cualquier tipo válido):

Entero	**int**
Booleano	**bool**
Puntero	***tipo**
Registro	**<tipo, tipo, ..., tipo>**
Array	**[tipo, entero]**

Para nuestro lenguaje dos arrays de un mismo tipo base, con el mismo número de elementos en total (independientemente de sus tamaños y dimensiones) son considerados tipos equivalentes. De hecho, un array de cualquier dimensión de un determinado tipo base es el mismo tipo que un puntero de este tipo base (vimos que considerar tipos distintos los arrays del mismo tipo base pero distinta dimensión o tamaño y no relacionarlos tampoco con los tipos puntero implicaba que no podríamos hacer funciones para ser invocadas por arrays de tamaño genérico, obentdríamos un error de tipo en la invocación). Sin embargo, un array de k elementos no se considera equivalente a un registro de k elementos de este tipo. Además, dos registros únicamente se consideran el mismo tipo si en su declaración tienen los mismos tipos y en el mismo orden.

3. EXPRESIONES

Nuestro lenguaje contempla expresiones aritméticas, lógicas y de tipo puntero (devolviendo direcciones de memoria). Se componen de constantes, variables, accesos a variables (en el caso de registros y arrays) además de los siguientes operadores:

OPERADORES	**ARIDAD**	**ASOCIATIVIDAD**	**INFIJO/PREFIJO**	**PRIORIDAD**
not, -, &	1	Sí (unario)	Prefijo	3
and, *, /	2	Sí, por la izquierda	Infijo	2
or, +, -	2	Sí, por la izquierda	Infijo	1
==, !=, >, <, <=, >=	2	No	Infijo	0

Los operadores se interpretan con su semántica usual y se pueden aplicar a los tipos en los que esta tiene sentido (not, and y or para operandos booleanos, & para obtener direcciones de memoria de variables al estilo C y los demás operadores en expresiones aritméticas). No se permite aplicar los operadores ==, != a expresiones no aritméticas.

4. INSTRUCCIONES

Nuestro lenguaje permite instrucciones de declaración, asignación, condicionales if, if-else, switch, bucles while y llamadas a funciones y procedimientos (y por tanto recursión). En concreto, las instrucciones soportadas y su semántica son:

- **Declaración:** Permite la creación de nuevas variables de cualquier tipo.

- **Declaración e inicialización:** Permite la creación de variables con una inicialización concreta. Sin embargo, nuestro lenguaje no permite inicializar en su declaración arrays o registros.
- **Asignación:** Permite asignar un valor determinado a una sección de memoria direccionable por parte de nuestro lenguaje (variables, elementos de arrays, direcciones apuntadas por un puntero, o un elemento de un registro) igualándola a una expresión válida. En nuestro lenguaje, el resultado de la expresión debe ser del mismo tipo que la dirección de memoria a asignar. Al igual que en la inicialización, no se soporta la inicialización de multiples elementos de registros ni arrays con una instrucción ni se proveen pseudoinstrucciones para este fin.
- **If e if-else:** Con su semántica habitual. La condición (una expresión booleana) debe ir entre paréntesis y es obligatorio el uso de corchetes en el bloque de instrucciones de if y else.
- **Switch:** Nuestro lenguaje soporta la instrucción switch de ejecución condicional en función únicamente del valor de una expresion aritmética (necesariamente entre paréntesis). Además, cada rama del switch debe contener un bloque de instrucciones entre corchetes y en la ejecución del switch se ejecuta a lo sumo uno de estos bloques, sin ser necesario (estando de hecho prohibido) el uso de break. Se permite el uso de default como última claúsula del switch para forzar la ejecución de un bloque de instrucciones por defecto.
- **While:** Con su semántica habitual. La condición debe estar incluida entre paréntesis. Y es obligatorio el uso de corchetes en el bloque de instrucciones.
- **Invocación a procedimientos y funciones:** Las llamadas a los procedimientos y funciones se realiza con el paso de parámetros por valor, en el caso de querer evitar esto es necesario el uso de punteros. De esta forma se garantiza que los atributos de un procedimiento o función mantienen su valor al terminar la ejecución este. Las funciones siempre tendrán por última instrucción el retorno de un valor de su mismo tipo (mediante el uso de la palabra reservada **return**). En el caso de llamadas a funciones si no se desea preservar su valor de retorno se les puede invocar como a los procedimientos, sin ser necesario que se encuentren en la parte derecha de una asignación.

La sintáxis de las instrucciones soportadas es la siguiente:

Declaración	**tipo identificador;**
Declaración con inicialización	**tipo identificador = expresión;**
Asignación	**identificador = expresión;**
If	**if(expB){ instrucciones }**
If-else	**if(expB){ instrucciones } else { inst }**
Switch	**switch(expA){** **case valor1: { inst }** **...** **case valorK: { inst }** **default: { inst }** **}**
While	**while(expB){ inst }**
Llamada a procedimiento/función	**identificador(id, ... , id);**

5. EJEMPLOS

5.1.- Requisitos mínimos:

Declaración de variables (tipado explícito)	int idVar;
Declaración de variables con inicialización	bool idVar = true;
Declaración de arrays (tipado explícito)	[int, 7] idArraySieteEnteros;
Declaración de arrays mutidimensionales	[[bool, 5],5] idMatrizCincoPorCinco;
Operadores booleanos (prioridad des.)	not > and > or
Operadores aritméticos (prioridad des.)	- > *,/ > +,-
Operadores relacionales	<,>,<=,>=,==,!=
Expresiones con constantes	(5*3 + 2)/2 not (true or false)
Expresiones con identificadores	4*x + 17 not (y1 and y2 or not y3)
Expresiones con elementos de arrays	4*x[5] + y[6][k] not (y1[j][k][l] or y2)
Expresiones con operadores infijos	4*x[5]+12 == y[i][a[i] + j*4] + aux*3
Instrucción de asignación	x = 5; y[12] = true; x[9] = y + 15*a[3];
Instrucción if	if (x == 5) {instrucciones}
Instrucción if-else	if (x == 5) {instrucciones} else {instrucciones}
Instrucción while	while(x==5) {instrucciones}

5.2.- Elementos opcionales:

Punteros	*int ptrId;
Punteros con inicialización	*bool ptrId = null; *int ptrId = &varId;
Obtención dirección de memoria de una variable	*int ptrId = &varId; (operador prefijo &)
Acceso al contenido de la posición de memoria que apunta el puntero	int varId = *ptrId; (operador prefijo *)
Procedimientos	void procId(int param1, bool param2) { inst. }
Funciones	int funcId([bool] array, int size) { … return val;}
Invocación procedimiento	procId(param1, … , paramN);
Invocación función	x = funcId(param1, … , paramN);

Instrucción switch	switch(x+1) {
	case 1: { print(“x es cero”); }
	case 2: { print(“x es uno”); }
	default: { print(“ni idea de qué es x”); }
	}

Registros	<int,[bool,7]> x; x<0> = 1; x<1>[0] = true;

6. EJEMPLO DE PROGRAMA

```
// Búsqueda binaria en un vector
int busqbin(*int vector, int iz, int de, int elem){
        int devuelve = -1;
        while(iz <= de){
                int m = iz + (de - iz) / 2;
                if (vector[m] == x){
                        devuelve = m;
                }else  {
                        if (vector[m] < x)  {
                                iz = m + 1 ;
                        }else { de = m - 1 ; }
                }
        }
        return devuelve ;
}

// Cálculo de el n-ésimo número de fibonacci en O(n)
int fibonacci(int n){
        <int,int> lasts;
        lasts<0>=1;
        lasts<1>=1;
        int aux ;

        while(n>=1){
                aux = lasts<0> + lasts<1>;
                lasts<0> = lasts<1>;
                lasts<1> = aux;
        }

        return lasts<1>;
}

// Procedimiento para inicializar una matriz n*m
void inizializaMatriz(*int matriz, int n, int m) {
        int i =0;
        int j =0;

        while(i<n){
                while(j<m){
                        matriz[i][j]=i+n*j;
                }
        }
}
```

```
// Función main
int main(){
	// Declaración e inicialización de un array unidimensional
	[int, SIZE] array;
	int k = 0;
	while(k<SIZE) { array[k] = k*k; k = k+1; }

	// Búsqueda binaria en el array (pasado como parámetro por referencia)
	int j = busqbin(array, 0, SIZE-1, 16);

	switch(j){
		case 0:   {print(“Es el primer elemento”);}
		case  -1: {print(“Elemento no encontrado”);}
		default:  {print(“Elemento encontrado”);}
	}

	// Inicialización y muestra de una matriz
	[[int, SIZE], SIZE] matriz;
	inizializaMatriz(matriz, SIZE, SIZE);

	int i =0;
	int j =0;
	while(i<SIZE){
		while(j<SIZE){
			print(matriz[i][j]);
		}
	}

	return 0;
}
```