# Procesadores de Lenguaje: Entrega 1

Marcos Brian Leiva y Pablo Villalobos

## Introducción, variables y tipos básicos

Nuestro lenguaje soporta 5 tipos básicos:

* Números enteros (Int). Ejemplos: 5, 17, -38512, inf
* Números reales (Dec). Ejemplos: 3.1416, -0.005
* Booleanos (Bool). Ejemplos: true, false
* Cadenas de caracteres (String). Ejemplos: “Hola”, ‘ejemplo’

Todos los identificadores de tipos empiezan con mayúscula. Las variables se declaran indicando su tipo y un identificador formado por letras y dígitos y comenzando en una letra minúscula.

Int numEnt;

String m817;

Como todas las instrucciones en nuestro lenguaje, las declaraciones de variables acaban en punto y coma. Las asignaciones consisten en un identificador, un igual, y un valor del tipo correspondiente.

numEnt = 180;

m817 = “Introduzca su nombre de usuario.”;

Se pueden unir declaración y asignación en la misma instrucción. No se puede declarar un identificador que ya ha sido declarado.

Bool exito = false;

Int exito = 0; -> Error!

Se pueden escribir comentarios mediante el carácter #. Todo lo que haya detrás de # en una línea será ignorado.

Bool valido = true;

# Bool valido = false;

## Expresiones y operadores

Los tipos básicos admiten una serie de operadores.

* Tipos numéricos (Int, Dec):
  + Suma: a + b
  + Resta: a - b
  + Producto: a \* b
  + División: a / b
  + Módulo: a mod b

Todos ellos son infijos, asocian por la derecha, y su prioridad es:

Suma = Resta < Producto = Division = Módulo

Producen un valor entero si sus operandos son enteros, y decimal  
 si al menos uno es decimal.

* Booleanos:
  + Not: not a
  + And: a and b
  + Or: a or b

Not es prefijo, and y or infijos y asocian por la derecha. La prioridad es:

Not < Or < And

* Comparaciones:
  + Igual: a == b
  + Mayor: a > b
  + Menor: a < b
  + Mayor o igual: a >= b
  + Menor o igual: a <= b

Todos ellos infijos y no asociativos.

* Strings:
  + Concatenación: a ++ b

Infijo, asocia por la derecha.

## Tipos

Además de los tipos básicos, se pueden construir otros tipos: arrays, tuplas, diccionarios y structs.

**Arrays:** Listas de elementos de un tipo, de longitud variable. Su tipo es [<tipo>], por ejemplo [Int] o [ [Bool] ]. Se pueden construir arrays como una lista de valores de <tipo> entre corchetes, y acceder a sus elementos con un índice entre corchetes, tanto para lectura como para escritura.

[Int] miArray = [1,2,3,4,5];

Int val = miArray[0]; # val = 1

val = miArray[4]; # val = 5

miArray[3] = 7; # miArray = [1,2,3,7,5]

Los arrays admiten el operador de concatenación ++:

[Int] miArray = [1,2,3] ++ [4,5,6];

# miArray = [1,2,3,4,5,6]

Asimismo, los Strings pueden ser indizados como un array:

String str = “Buenos días”;

String caracter = str[0]; # caracter = “B”

**Tuplas**: Listas ordenadas de elementos de varios tipos de longitud fija. Su tipo es (<tipo1>, <tipo2>,...), por ejemplo (Dec, Dec). Se pueden acceder del mismo modo que un array.

(Dec, Dec) posicion = (1, 2.5);

Dec y = posicion[1]; # y = 2.5

**Diccionarios**: Asocian elementos de un tipo (clave) a elementos de otro (valor). Su tipo es [<tipo1>:<tipo2>], por ejemplo [String:Int]. Se pueden construir diccionarios como listas de pares clave-valor, y acceder al valor de cada clave mediante corchetes para lectura, escritura e inserción.

[String:Int] edades = [ “manuel” : 25, “maria” : 30 ];

Int edadM = edades[“maria”]; # edadM = 30

edades[“manuel”] = 26; # edades = [ “manuel” : 26, “maria” : 30 ]

edades[“pepe”] = 54; # edades = [ “manuel” : 26, “maria” : 30, “pepe” : 54 ]

**Structs**: Agrupaciones de un conjunto fijo de valores con nombre (campos). Su tipo es ‘struct [ <nombre1> : <tipo1>, <nombre2> : <tipo2>, ...]’, por ejemplo ‘struct [ nombre: String, altura: Dec]’. Se puede acceder a cada campo mediante el operador punto.

struct [ identificador: String, valido: Bool] autorización;

autorizacion.identificador = “4338”;

autorizacion.valido = true;

**Punteros**: Almacenan una referencia a un valor de cierto tipo. Su tipo es ^<tipo>, por ejemplo ^Int. Se puede acceder al valor referenciado mediante el operador de desreferencia, ^, y dado un valor almacenado en una variable o estructura, se puede obtener un puntero a él mediante el operador de referencia, &.

Int val1 = 3;

^Int punt = &val1; # ^punt = val1 = 3

^punt = 7; # ^punt = val1 = 7

struct [ valor: Int ] miStruct;

miStruct.valor = 15;

punt = &miStruct.valor; # ^punt = 15

**Tipos con nombre**: El usuario puede definir nuevos tipos que sean combinación de los anteriores y darles un nombre. Esto se hace mediante la instrucción type:

type <IdenTipo> = <tipo>;

El nombre del nuevo tipo puede contener letras y digitos, y debe comenzar por una letra mayuscula. Por ejemplo:

type Persona = struct [ nombre: String, apellidos: String, edad: Int, hobbies: [String] ];

Persona per1;

per1.nombre = “Lola”;

**Null**: El valor null resulta al intentar acceder a un elemento no existente. El valor de una variable sin inicializar es null, y acceder a un array o tupla fuera de sus límites resulta en null. Acceder a una clave de un diccionario que no tiene valor asociado también produce null, así como desreferenciar un puntero que no apunta a nada.

[Int] lista1 = [1,2,3];

Int val; # val = null

val = lista1[6]; # val = null

^Int punt;

{

Int i = 3;

punt = &i; # ^punt = 3

}

# ^punt = null;

El valor null es el único valor del tipo Null, pero se puede interpretar como de cualquier otro tipo.

Int vacio = null;

String vacio2 = null;

Null varNull = 6; -> Error!

El valor null solo admite el operador ==. Cualquier otro operador que reciba null como operando producirá un error.

Int i1 = 10;

Int i2;

Bool b1 = i1 == null; # b1 = false

Bool b2 = i2 == null; # b2 = true

Int i3 = i1 + i2; -> Error!

## Instrucciones

Además de las instrucciones ya vistas: declaración de variables, asignación, y definición de tipos, existen las siguientes:

**Bloque**: Es un subprograma entre llaves con un ámbito propio. Las instrucciones dentro del bloque pueden utilizar las variables, tipos, etc definidas anteriormente, pero cualquier declaración dentro del bloque se pierde tras ejecutar la última instrucción.

Int valor1 = 10;

{

Int valor2 = 2\*valor1;

valor1 = 0;

};

# valor1 = 0. No existe valor2.

**If**: Condiciona la ejecución de un bloque a una expresión booleana. Si la expresión tiene valor True, se ejecuta el bloque. Si no, se ignora.

if valor1 > 0 {

valor1 = valor1-1;

};

Se puede añadir una cláusula **else** después del bloque, que se ejecuta en caso de que la condición sea false.

if valor1 > 0 {

valor1 = valor1-1;

} else {

valor1 = 100;

};

**Case**: Es similar a un if, pero evalúa expresiones de tipo arbitrario, y permite definir ramas de ejecución según el valor de la expresión. Cada rama se define como <exp> -> <bloque>. Exp es una expresión del tipo adecuado, o puede ser ‘default’, que admite cualquier valor distinto de los especificados en las otras ramas. Si el valor de la expresión de alguna rama coincide con el de la expresión inicial, se ejecuta el bloque de esa rama. Si no, se ignora la instrucción.

Int responseCode = 404;

Int serverErrorCode = 501;

case responseCode {

200 -> { result = “OK”; }

404 -> { result = “Error: not found”; }

serverErrorCode -> { result = “Server error”; }

default -> { result = “Unknown error”; }

};

# result = “Error: not found”

**Repeat**: La instrucción repeat es un bucle. En su forma más básica, repite un bloque de código un número de veces especificado por una expresión: ‘repeat <exp> <bloque>’. Si exp es inf, el bucle nunca termina.

Int contador = 2;

Int reps = 5;

repeat 2\*reps {

contador = contador\*2;

};

# contador = 2048;

Repeat tiene otra variante, de la forma ‘repeat <exp1> | <exp2> <bloque>. Exp1, como antes, es una expresión de tipo Int que indica un número de repeticiones. Exp2 es una condición booleana. En esta variante, el bloque se repite como máximo exp1 veces. En cada repetición se evalúa exp2, y si su valor es false, se detiene la ejecución del bucle.

Int contador = 2;

repeat 200 | contador < 1000 {

contador = contador\*2;

}

# contador = 1024

El resto de instrucciones están relacionadas con las funciones.

## Funciones

Las funciones son bloques de código con nombre y una lista de parámetros, que se pueden ejecutar en cualquier momento llamándolas de la siguiente forma:

<idenFun> (<par1>, <par2>,... );

idenFun es un nombre de función, que tiene la misma forma que un nombre de variable, y los parámetros son expresiones del tipo que requiera la función. El ejemplo más simple es la función print, que admite un parámetro de tipo String y escribe este parámetro por la salida estándar.

print(“Hello world!”);

Las funciones devuelven un valor. Si se llama a una función sin más, el valor de retorno es ignorado. Pero también puede usarse en una expresión o asignarse a una variable. Por ejemplo, la función len admite como parámetro una lista, tupla o diccionario y devuelve el número de elementos que contiene.

[Int] lista = [6,12,45,7,193];

Int mitad = len(lista) / 2; # mitad = 2

El usuario puede definir funciones propias de la siguiente forma:

fun <tipo> <idenFun> ( <iden1> : <tipo1>, <iden2>: <tipo2>,... ) = <bloque>;

Tipo es el tipo del valor que devuelve la función, idenFun es su nombre, y entre paréntesis hay una lista de parámetros con sus tipos. Los identificadores de los parámetros se pueden usar como variables dentro del bloque de la función, y tendrán los valores con los que se ha hecho la llamada.

En el bloque debe haber una instrucción return, de la forma ‘return <exp>;’ que indique el valor devuelto por la función. La instrucción return puede ejecutarse el cualquier momento para detener la ejecución de la función.

fun Dec potencia (base: Dec, exp: Int) = {

Dec res = 1;

repeat exp {

res = res \* base;

};

return res;

};

Si no se espera que una función devuelva un valor útil, se puede definir como de tipo Null, y utilizar ‘return null’. En este caso se puede omitir la instrucción return, y la función terminará su ejecución cuando termine la de su bloque.

fun Null saludar (nombre: String) = {

print(“Hola “ ++ nombre ++ “!”);

}

## Funciones predefinidas

Null print(String): escribe por la salida estándar.

String read(): lee por la entrada estándar.

Null push(Array): Añade un elemento al final de un array.

Null pop(Array): Elimina el último elemento de un array.

Lista keys(Dict): Devuelve un array de claves de un diccionario.