UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Procesadores de Lenguaje:

Nombre de nuestro lenguaje

Curso 2018 / 2019

invalid utf-8 byte sequence spanish characters listing latex

Realizado por:
Medina Medina, David Alberto
Brito Ramos, Christian
Hernández Delgado, Christopher
López González, Néstor

Índice

1. Definición del lenguaje (Autor: Quien termine antes)

Introducir breve introducción del lenguaje que planteamos.

1.1. Tipos de datos (David)

Cualquier leguaje de programación necesita definir un conjunto de *tipos de datos*, esto es, la batería de valores y operaciones que puede adquirir una variable. Cada tipo de dato está definido en el lenguaje por un *literal* único que lo representa, lo que permite que cada tipo de dato tenga un representación física específica.
Los tipos de datos definidos en el lenguaje son los que figuran en el *cuadro 1*. Las características críticas de implementación que define a cada tipo son:

Entero Representa a todas y cada una de las variables enteras que sean declaradas en el lenguaje. Este tipo de dato presenta un tamaño de 4 bytes (32 bits) y permite representar números enteros con signo. El *complemento a 2* es el sistema elegido para definir el signo del número entro. Este dato se representa por el literal `int`. El rango de valores que puede tomar es

$$\left[-2^{N-1},\, 2^{N-1}-1\right] = \left[-2^{32-1},\, 2^{32-1}-1\right] = [-2147483648,\, 2147483647] \tag{1}$$

donde, N es el número de bits disponibles para representar el número el número entero (32 bits).

Coma flotante de simple precisión Este tipo de dato representa número reales en coma flotante de simple precisión con un tamaño de 4 bytes (32 bits) siguiendo el estándar *IEEE 754*. En la figura 1.1 puede observarse como esta representación binaria los bits se organizan en tres sectores principales:

- **Signo** (1 bit). Se trata de un sólo bit que define el signo del número: positivo (0) o negativo (1).
- **Exponente (8bits)**. Se trata de un número entero con signo de complemento a 2 ($[-128,\, 127]$)
- **Mantisa (23 bits)**. Conforma la fracción a la derecha de la coma binaria y un bit de encabezado implícito.

Este tipo de dato se representa con el literal `real`. Su rango de valores es de $\left[1{,}18 \cdot 10^{\check{}38},\, 3{,}4 \cdot 10^{38}\right]$.

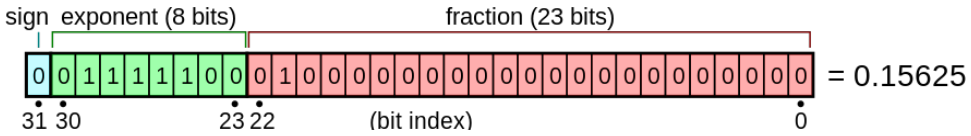

Figura 1: Representación binaria de número en coma flotante de simple precisión (*IEEE 754*)

Caracter Este tipo de dato es usado para representar caracteres de codificación *UTF-8*, es por este motivo que el tamaño de que ocupan las variables de tipo caracter son de 4 bytes siendo `char` el literal que lo representa.

Booleano Se trata de un tipo de dato utilizado para representar representar valores booleanos. Su tamaño es de 1 bit, por lo que tan solo puede tomar dos valores: `1` (verdadero) ó `0` (falso). El literal que lo representa es `bool`.

Cuadro 1: Tipos de datos

Tipo	**Literal**	**Tamaño**	**Rango**
Entero	int	4 Bytes	$[-2147483648,\ 2147483647]$
Coma flotante de simple precisión	real	4 Byte	$\left[1{,}18 \cdot 10^{\check{}38},\ 3{,}4 \cdot 10^{38}\right]$
Caracter	char	1 Byte	*UTF-8*
Lógico	bool	1 Bit	$[0,\ 1]$

En el siguiente ejemplo se muestra cómo se declaran las variables con los literales de los tipos descritos anteriormente:

```
int entero      .. Esto es un entero
real flotante   .. Esto es un numero en coma flotante
char caracter   .. Esto es un caracter
bool booleano   .. Esto es un booleano
```

1.2. Colecciones de datos: `Arrays`

Las variables pueden ser agrupadas en colecciones de datos de una dimensión denominados *arrays*. En este lenguaje, cualquier tipo de dato puede formar parte de un *array*.

1.3. Palabras reservadas (Christian)

Aquí va el texto. Poner siempre un código de ejemplo.

1.4. Comentarios (Christian)

Aquí va el texto. Poner siempre un código de ejemplo.

1.5. Tipos de operadores

Aquí va el texto. Poner siempre un código de ejemplo.

1.5.1. Operadores aritméticos (David)

Aquí va el texto. Poner siempre un código de ejemplo.

1.5.2. Operadores lógicos (Christopher)

Aquí va el texto. Poner siempre un código de ejemplo.

1.5.3. Operadores bit a bit (Christian)

Aquí va el texto. Poner siempre un código de ejemplo.

1.5.4. Operadores de array (Néstor)

Los operadores de arrays nos permiten realizar operaciones con arrays, tratándolos como a subconjuntos. Los operadores a definir son los siguientes:

Unión La unión de dos arrays consiste en crear un nuevo array formado por los elementos del primer array y los del segundo.

Diferencia La diferencia de un array con otro consiste en crear un nuevo array con todos los elementos del primer array, descartando aquellos que aparecen en el segundo.

Intersección La intersección de dos arrays consiste en crear un nuevo array formado por todos los elementos comunes entre el primer y el segundo array.

Concatenación La Nuevo array formado por los elementos del primero seguido de los del segundo.

Cuadro 2: Tipos de operadores de arrays

Operador	Símbolo	Formato
UNIÓN	U	{ARR} U {ARR}
DIFERENCIA	D	{ARR} D {ARR}
INTERSECCIÓN	I	{ARR} I {ARR}
CONCATENACIÓN	+	{ARR} + {ARR}

A continuación mostramos un ejemplo de uso de los operadores de array para un caso trivial meramente ejemplificativo:

```
int [6]  a = new int [6]{0,1,2,3,4,5}
int [7]  b = new int [7]{3,4,5,6,7,8,9}

int []  c = a U b ..c =  {0,1,2,3,4,5,6,7,8,9}
int []  d = a D b ..d = {0,1,2}
int []  e = a I b  .. e = {3,4,5}
int []  f = (a U b) D (a I b)  .. f =  {0,1,2,6,7,8,9}
int []  g = d + f + a ..g =   {0,1,2,0,1,2,6,7,8,9,0,1,2,3,4,5}
```

1.6. Estructuras de control

Aquí va el texto. Poner siempre un código de ejemplo.

1.6.1. Sentencias `if-ifelse-else` (Néstor)

La estructura de control if-ifelse-else sentencia condicional que está compuesta de los siguientes bloques:

Bloque `if` Este bloque está identificado por la siguiente estructura:

$$\{Expr\}\ ?: \tag{2}$$

En caso de ser verdadera, se ejecutarán las instrucciones que van seguidas de la expresióm 4. Sólo puede existir un único bloque `if` al principio de una sentencia `if-ifelse-else`.

Bloque `ifelse` Si no se cumple la condición del bloque anterior, se comprueba si la condición de este bloque se cumple para posteriormente ejecutar el conjunto de instrucciones que se encuentran en su contexto. En caso contrario, pasa a ejecutarse el siguiente bloque `elseif` La estructura de este bloque es el siguiente:

$$.\ \{Expr\}\ ?: \tag{3}$$

Bloque `else` Si ninguna de la expresiones de los bloques anteriores cumple la condición, se ejecutarán las instrucciones contenidas en este bloque. La estructura de este bloque es la siguiente:

$$.?: \tag{4}$$

Sólo puede existir un único bloque `else` al final de una sentencia `if-ifelse-else`.

Listing 1: Ejemplo de uso de la sentencia `if-ifelse-else`

```
int a = 25
int b = 0

a > 20 ?:
    b = a - 10
. a < 10 ?:
    b = a + 5
.?:
    b = -1
```

1.6.2. Bucle `for-forelse-else` (Christopher)

Aquí va el texto. Poner siempre un código de ejemplo.

1.6.3. Bucle `while-whileelse-else` (Christian)

Aquí va el texto. Poner siempre un código de ejemplo.

1.7. Funciones (David)

Aquí va el texto. Poner siempre un código de ejemplo.

1.8. Funciones primitivas (Néstor)

En nuestro lenguaje, Van a existir predefinidas un conjunto de funciones primitivas que facilitarán al programador su trabajo en el desarrollo del programa. Las funciones están divididas en las siguientes categorías:

Interacción entrada-salida Las funciones básicas de E/S son:

`bool print(T salida)` Esta función permite mostrar por pantalla el valor de `salida`, cuyo tipo `T` puede ser `int, real, bool, char` o una string (`char[]`). Si la operación termina correctamente la primitiva devolverá `true`, en caso contrario, devuelve `false`.

`bool scan(char[] entrada)` Esta rutina permite leer el valor pasado por teclado una vez se pulse `ENTER` (`0x0D`). Si la operación termina satisfactoriamente la primitiva devolverá `true`, en caso contrario, devuelve `false`.

Listing 2: Ejemplo de uso de la función `print()` y `scan()`

```
int number
char[] name
print("Introduzca el primer número: ")
a = scan()
print("¿Cuál es su nombre? ")
name = scan()
print("El usuario " + name + " ha introducido el número " + number)
```

Funciones para arrays numéricos Las funciones para arrays numéricos permiten operaciones que impliquen el uso de arrays de enteros, números en coma flotante o combinar ambos tipos. El tipo `T` de las siguientes funciones indica que el tipo del parámetro de entrada o el tipo de retorno de la función puede ser `int` o `real`:

`int length(array)` Devuelve el número de elementos del array.

`T sum(array)` Devuelve la suma de todos los elementos contenidos en el array.

`T res(T[] array)` Devuelve el resultado de restar todos los elementos del array.

`real prod(T[] array)` Devuelve el producto de todos los elementos contenidos en el array.

`int prod(T[] array1, T[] array2)` Devuelve el producto escalar de los dos arrays.

`real coc(T[] array)` Devuelve el cociente de todos los elementos contenidos en el array.

`T[] append(T[] array, T[] element, int position)` Coloca en la posición `position` del array un elemento que debe ser del mismo tipo que el resto del array. Los demás elementos a partir de la posición especificada serán desplazados una posición a la derecha. Se devuelve la referencia al array resultante.

`T[] delete(T[] array, int position)` Elimina el elemento de la posición `position` del array y devuelve la referencia al array resultante. Los demás elementos a partir de la posición especificada serán desplazados una posición a la izquierda.

`T[] find(array,element)` Devuelve una referencia a un array de enteros, que representan las posiciones donde se encuentra un elemento en el array.

`T[] sort(T[] array, T[] ascendent)` Devuelve la referencia a un array con los elementos ordenados de forma ascendente si `ascendent` es igual a `true`; en caso contrario, los ordenará de forma descendente.

`bool equal(T[] array1, T[] array2)` Devuelve `true` si los dos arrays son iguales en tamaño y los elementos están dispuestos en el mismo orden. En caso contrario, devuelve `false`.

Listing 3: Ejemplo de uso de funciones primitivas para arrays numéricos

```
int [] v1 = {2,5,9,5}
int [] v2 = {5,4,8,9,3}
int tam = length(v1)      .. tam = 4
int n1 = sum(v1)          .. n1 = 21
int n2 = sum(v2)          .. n2 = 29
int [] v3 = sum(v1,v2)      .. v3 = {7,9,17,14,3}
int [] v4 = res(v1,v2)      .. v4 = {-3,1,1,-4,-3}
int n3 = prod(v1)         .. n3 = 450
int n4 = prod(v2)         .. n4 = 4320
int [] v5 = prod(v1,v2)     .. v5 = {10,20,72,45,0}
int [] v6 = coc(v3,v2)      .. v6 = {2,5,9,5,0}
float [] v7 = coc(v1,v2)    .. v7 = {0.4,1.25,1.125,0.5555555556,0.0}
append(v1,6,3)            .. v1 = {2,5,6,9,5}
delete(v1,3)              .. v1 = {2,5,9,5}
int [] found1 = find(v1,5)  .. found1 = {2,4}
int [] found2 = find(v1,8)  .. found2 = {}
sort(v2,true)             .. v2 = {3,4,5,8,9}
sort(v2,false)            .. v2 = {9,8,5,4,3}
boolean b1 = equal(v2,v3) .. b1 = false
boolean b2 = equal(v1,v1) .. b2 = true
```

Funciones para strings Las funciones para strings permiten realizar operaciones sobre strings. Las funciones principales son:

`int length(char[] string)` Devuelve el número de caracteres de la string.

`char[] substring(char[] string, int position)` Devuelve la referencia a la substring que se encuentra entre `position` y el final de `string`.

`char[] substring(char[] string, int beginning, int end)` Devuelve la referencia a un substring ubicado entre la posición `beginning` y la posición anterior al índice `end`.

`char[] trim(char[] string)` Elimina todos los espacios (`0x20`) de la string.

`bool equal(char[] string1, char[] string2)` Devuelve `true` si las dos strings son iguales en tamaño y los caracteres que lo conforman están dispuestos en el mismo orden. En caso contrario, devuelve `false`.

Listing 4: Ejemplo de uso de funciones primitivas para strings

```
char[] s1 = "Hola"
char[] s2 = " mundo"
char[] s3 = s1 + s2                    .. s3 = "Hola mundo"
int tam1 = length(s1)                  .. tam1 = 4
int tam2 = length(s3)                  .. tam1 = 10
char[] s4 = substring(s3,6)               .. s4 = "mundo"
char[] s5 = substring(s3,4,8)          .. s5 = "a mu"
char[] s6 = " " + s1 + " " + s2 + "    " .. s6 = " Hola   mundo   "
int tam3 = length(s6)                  .. tam3 = 17
trim(s6)                            .. s6 = "Hola mundo"
int tam4 = length(s6)                  .. tam4 = 10
boolean b1 = equal(s3,s6)              .. b1 = true
boolean b2 = equal(s1,s3)              .. b2 = false
```

1.9. Código ejemplo (Christopher)

Aquí va el código de ejemplo con el que probaremos nuestro compilador.