



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

PROCESADORES DE LENGUAJE:

Nombre de nuestro lenguaje

Curso 2018 / 2019

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



Índice

1. Definición del lenguaje (Autor: Quien termine antes)	2
1.1. Tipos de datos (David)	3
1.2. Colecciones de datos: Arrays	4
1.3. Palabras reservadas (Christian)	6
1.4. Comentarios (Christian)	7
1.5. Tipos de operadores	8
1.5.1. Operadores aritméticos (David)	8
1.5.2. Operadores lógicos (Néstor)	8
1.5.3. Operadores bit a bit (David)	8
1.5.4. Operadores de array (Christopher)	8
1.6. Estructuras de control	9
1.6.1. Sentencias if-ifelse-else (Néstor)	9
1.6.2. Bucle for-forelse-else (Christopher)	9
1.6.3. Bucle while-whileelse-else (Christian)	9
1.7. Funciones (David)	10
1.8. Funciones primitivas (Néstor)	11
1.9. Código ejemplo (Christopher)	12

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

Introducir breve introducción del lenguaje que planteamos.

1.1. Tipos de datos (David)

Cualquier lenguaje 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 una 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 entero. Este dato se representa por el literal `int`. El rango de valores que puede tomar es

$$[-2^{N-1}, 2^{N-1} - 1] = [-2^{32-1}, 2^{32-1} - 1] = [-2147483648, 2147483647] \quad (1)$$

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

Coma flotante de simple precisión Este tipo de dato representa números 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 cómo 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 (8 bits)**. Se trata de un número entero con signo de complemento a 2 ($[-128, 127]$)
- **Mantisa (23 bits)**. Conforman 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 $[1,18 \cdot 10^{-38}, 3,4 \cdot 10^{38}]$.

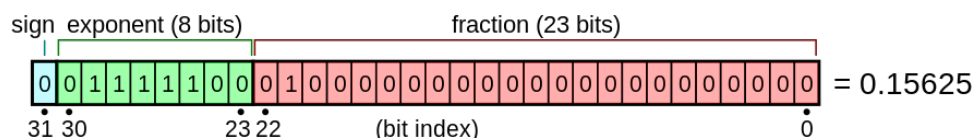


Figura 1: Representación binaria de número en coma flotante de simple precisión (*IEEE 754*)

Caracter Este tipo de dato es usado para representar caracteres con la codificación *ASCII*, es por este motivo que el tamaño de que ocupan las variables de tipo caracter son de 1 byte de tamaño, siendo `char` el literal que lo representa.

Booleano Se trata de un tipo de dato utilizado para 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	$[1,18 \cdot 10^{-38}, 3,4 \cdot 10^{38}]$
Caracter	char	1 Byte	$[0x00 - 0xFF]$
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**. Para declarar un **array** del tipo que se desee, debe usar la gramática 2:

$$tipo[< int >] nombre_variable \quad (2)$$

donde el *tipo* define el tipo de dato e $< int >$ un valor entero opcional que define el tamaño del **array**. Cuando no se indica este último valor entero, no se lleva a término la reserva en memoria del **array** declarado. En caso contrario, se reservará en memoria tantos bytes/bits como fueren necesarios para generar una colección de tipos de datos del tamaño indicado por $< int >$. El número de bytes/bits a reservar está determinado por el tipo de dato y el tamaño del **array**,

$$Tamano_{memoria} = Tamano_{tipo\ dato} \times Tamano_{array} \quad (3)$$

Un **array** que ha sido declarado con anterioridad puede ser redefinido haciendo uso de la gramática 4,

$$tipo[] nombre_variable = new\ tipo[< int >] \{expresion_1, expresion_2, \dots\} \quad (4)$$

Esta notación es alternativa a la gramática 2, donde podremos inicializar el **array** a un conjunto de expresiones separados por comas encerrados dentro de los caracteres $\{ y \}$. Estas expresiones son opcionales. El tamaño máximo del vector y, por tanto, de expresiones es el indicado por $< int >$ el cual es un número entero de carácter obligatorio.

Si se declara un **array** utilizando la gramática 2 o 4 sin expresiones en esta última, el **array** es inicializado en todas sus posiciones al valor 0 para tipos de datos enteros y reales. Para caracteres el valor por defecto es el carácter nulo. Y para tipos booleanos el valor por defecto es **FALSE**.

La gramática 5 es necesaria para acceder al valor de un elemento del **array** en una posición arbitraria del mismo,

$$\text{nombre_variable} [< \text{int} >] \quad (5)$$

donde $< \text{int} >$ es un entero obligatorio que indica la posición del **array** a la que se desea acceder.

El **array** de caracteres constituyen los denominados **string**, los cuales requieren una atención especial ya que es posible cargar una variable con una secuencia de caracteres sin ser necesaria la declaración dada por la gramática 4. La gramática 6 define la instancia de un **string** con un conjunto de caracteres localizados entre los caracteres comillas doble,

$$\text{nombre_variable} = " < \text{char} > < \text{char} > \dots" \quad (6)$$

El siguiente listado muestra algunos ejemplos de uso de los **array** definidos en este lenguaje:

```

..
    En este bloque se declara un array de cada tipo de variable sin asignacion de espacio en
        memoria
..

int [] entero
real [] real
char [] string
bool [] booleano

..
    En este bloque se declara un array de cada tipo con asignacion de espacio en memoria (de
        tamano 5 cada uno)
..

int [5] entero5
real [5] real5
char [5] string5
bool [5] booleano5

..
    En este bloque se declara un array de cada tipo y se instancia cada uno de los valores de los
        elementos que lo componen
..

int [] entero2 = new int[2]{ 0, 1}
real [] real2 = new real[2]{0.0, 1.0}
char [] string2_ab = "ab" .. Declaracion con asignacion alternativo al tipo string
char [] string2_cd = new char[2]{'c', 'd'}
bool [] booleano2 = new bool[2]{true, false}

```

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)

Estos operadores son necesarios para realizar operaciones matemáticas sencillas. La descripción detallada de cada uno de estos operadores queda reflejado en la tabla 2.

Cuadro 2: Operadores aritméticos

Operación	Tipo operación	Lexema	Estructura	Tipo entrada	Tipo salida ¹
Suma	Binaria	+	{Expr} + {Expr}	int/real	int/real
Resta	Binaria	-	{Expr} - {Expr}	int/real	int/real
Multipliación	Binaria	*	{Expr} * {Expr}	int/real	int/real
División	Binaria	/	{Expr} / {Expr}	int/real	int/real
Módulo	Binaria	%	{Expr} % {Expr}	int/real	int/real
Potencia	Binaria	**	{Expr} ** {Expr}	int/real	int/real
Raíz	Binaria	#	{Expr} # {Expr}	int/real	int/real
Pre-incremento ²	Unaria	++	++{Expr}	int/real	int/real
Pre-decremento ²	Unaria	--	--{Expr}	int/real	int/real
Post-incremento ³	Unaria	++	{Expr}++	int/real	int/real
Post-decremento ³	Unaria	--	{Expr}--	int/real	int/real

¹ Todas las operaciones binarias permiten entremezclar el uso de un valor entero y otro real al mismo tiempo aunque esto conllevaría el *casteo* del resultado de la operación a un valor de tipo real.

² El valor de la variable es incrementado/decrementado una unidad previa asignación del mismo.

³ El valor de la variable es incrementada/decrementada una unidad después de asignar el valor del misma.

1.5.2. Operadores lógicos (Néstor)

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

1.5.3. Operadores bit a bit (David)

Estos operadores permiten manipular los valores de una expresión a nivel de bits. El cuadro 3 refleja la descripción detallada de cada uno de estos operadores.

1.5.4. Operadores de array (Christopher)

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

Cuadro 3: Operadores bit a bit

Operación	Tipo operación	Lexema	Estructura	Tipo entrada	Tipo salida
AND	Binaria	&	{Expr} & {Expr}	Cualquiera	Cualquiera
OR	Binaria		{Expr} {Expr}	Cualquiera	Cualquiera
XOR	Binaria	^	{Expr} ^ {Expr}	Cualquiera	Cualquiera
Desp. izq.	Unaria	<<	{Expr} << {Expr} ¹	Cualquiera	Cualquiera
Desp. dcha.	Unaria	>>	{Expr} >> {Expr} ¹	Cualquiera	Cualquiera

¹ Rota los bits de la expresión de la izquierda tantas unidades a la izquierda/derecha como indique la expresión de la derecha.

1.6. Estructuras de control

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

1.6.1. Sentencias if-otherwise-else (Néstor)

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

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)

Las funciones componen una estructura fundamental en cualquier lenguaje de programación que son útiles para estructurar funcionalmente el código en diferentes contextos, aportando una mayor expresividad del lenguaje.

Una función puede ser declarada sin especificar el código que se ejecutará dentro de su contexto. La forma general de declaración puede observarse en el listado 1.

Listing 1: Declaracion de funciones

```
fun {tipo} {nombre_funcion} ({parametro1}, [{parametro2}], ...)
```

La definición de cualquier función sigue la estructura del listado 2. El cuerpo de la función debe comenzar por el caracter de cambio de contexto (caracter 0x09 o cuatro caracteres 0x20 seguidos).

Listing 2: Declaracion de funciones

```
fun {tipo} {nombre_funcion} ({parametro1}, [{parametro2}], ...)
    {instruccion1}
    [{instruccion2}]
    ...
```

El listado 3 ilustra la estructura general de llamada a una función.

Listing 3: Declaracion de funciones

```
{nombre_funcion}({parametro1}, [{parametro2}], ...)
```

Es importante recalcar que la referencia en memoria de un array es leída cuando este es usado como parámetro de una función. Si un array es el valor de retorno de una función, se devuelve su referencia en memoria.

1.8. Funciones primitivas (Néstor)

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

1.9. Código ejemplo (Christopher)

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