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)

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 entero. 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 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 $\left[1{,}18 \cdot 10^{-38},\ 3{,}4 \cdot 10^{38}\right]$.

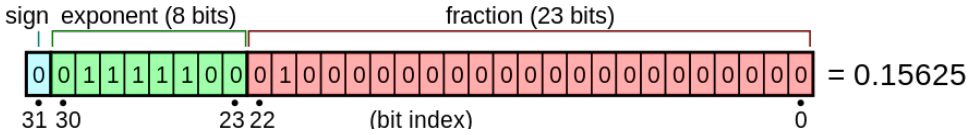

Figura 1: Representación binaria de número en coma flotante de simple precisión (*IEEE 754*)

Caracter Este tipo de dato es usado para representar caracteres 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 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:

$$tipo[< int >]\ nombre\_variable \tag{2}$$

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**,

$$Tamano_{memoria} = Tamano_{tipo\,dato} \times Tamano_{array} \tag{3}$$

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

$$tipo[\,]\ nombre\_variable = new\ tipo[< int >]\ \{expresion\_1, expresion\_2, ...\ \} \tag{4}$$

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 caracter 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 caracter **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,

$$nombre\_variable\ [< int >] \tag{5}$$

donde $< int >$ 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,

$$nombre\_variable\ = " < char >< char > ..." \tag{6}$$

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

Listing 1: Ejemplo de uso de arrays

```
,.
   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)

Este lenguaje utiliza una serie de palabras reservadas que se utilizan para desempeñar las funciones aquí descritas:

- **'continue'** . La sentencia de continue es de tipo de control de bucles. Dentro de la iteracion en un bucle, de cualquiera de los tipos (while, do-while, for), el uso de esta sentencia rompe la iteracion de dicho bucle. Provocando que se ejecute la siguiente iteracion de dicho bucle, ignorando las sentencias posteriores a çontinue".
- **'break'** . Dentro de la iteracion en un bucle, de cualquiera de los tipos (while, do-while, for), el uso de esta sentencia rompe la iteracion de dicho bucle.
- **'return'** . Palabra empleada para retornar el resultado de un método o función, además de interrumpir la ejecución del mismo.
- **'fun'** . Palabra especial utilizada al principio de cada declaración o definición de funciones.

1.4. Comentarios (Christian)

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

- **'Comentario de línea'** . Para indicar un comentario de línea, basta con comenzar con '..', siendo el resultado final '.. TEXTO'.

```
int a = 7
.. Variable que almacena una suma.
int suma = 0
suma = 30 + a
```

- **'Comentario de bloque'** . Los comentarios de varias líneas se establecen utilizando ',.' y '.,' quedando como resultado ',. VARIAS LÍNEAS DE TEXTO .,'.

```
int a = 7
int suma = 0
suma = 30 + a
,.
En este punto, la variable suma toma el
valor de 37.
.,
```

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
Multiplicación	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 (Christopher)

Definimos los símbolos que identificarán a los operadores lógicos de forma análoga a los utilizados por gran parte de otros lenguajes de programación, separando cada operador 2 expresiones a comparar por los operadores lógicos a utilizar, con la excepción del propio *NOT*, que podrá utilizarse directamente para determinar si el valor devuelto por una expresión de tipo `booleano` es directamente verdadero o no. Como dato adicional, también consideramos la posibilidad de hacer uso de comparadores mediante dichos operadores lógicos n-arios, comparando pares de expresiones entre sí de forma sucesiva siguiendo el orden de lectura de izquierda a derecha (que a efectos prácticos es lo mismo que hacer sucesivos *AND* pero de forma más breve) y la posibilidad de prioridades para ciertos operadores (como el *AND*).

En relación a las expresiones de comparación, consideramos hacer que de momento solo se puedan utilizar entre tipos de datos del mismo tipo o similar (por ejemplo, entre enteros y reales podría hacerse). Dejamos la tabla de operadores como sigue:

Cuadro 3: Tipos de operadores lógicos

Operador	**Símbolo**	**Formato**	**Tipos de datos comparables**
AND	&&	{EXPR} && {EXPR}	LOGICO - LOGICO
OR	||	{EXPR} || {EXPR}	LOGICO - LOGICO
NOT	!	!{EXPR}	LOGICO
IGUAL	==	{EXPR} == {EXPR}	LOGICO - LOGICO ENTERO - ENTERO ENTERO - REAL REAL - REAL CHAR - CHAR ARRAY[CHAR] - ARRAY[CHAR]
NOIGUAL	!=	{EXPR} != {EXPR}	LOGICO - LOGICO ENTERO - ENTERO ENTERO - REAL REAL - REAL CHAR - CHAR ARRAY[CHAR] - ARRAY[CHAR]
MAYOR	>	{EXPR} > {EXPR}	ENTERO - ENTERO ENTERO - REAL REAL - REAL
MAYORIGUAL	>=	{EXPR} >= {EXPR}	ENTERO - ENTERO ENTERO - REAL REAL - REAL

MENOR	<	{EXPR} < {EXPR}	ENTERO - ENTERO ENTERO - REAL REAL - REAL
MENORIGUAL	<=	{EXPR} <= {EXPR}	ENTERO - ENTERO ENTERO - REAL REAL - REAL

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

```
int a = 7
int suma = 0
5 <= a < 10 ?:
    suma = 4 * a

suma > 20 > a && 2*suma != 32 < a
    suma++
```

1.5.3. Operadores bit a bit (David)

Estos operadores permiten manipular los valores de una expresión a nivel de bits. El cuadro 4 refleja la descripción detallada de cada uno de estos operadores.

Cuadro 4: 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.

1.5.4. Operadores de array (Néstor)

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

1.6. Estructuras de control

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

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

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

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

La estructura de control evaluará un conjunto de expresiones hasta alcanzar los símbolos que hemos utilizado para definir bucles, el "`??`". Posteriormente se puede especificar una expresión que se usará comúnmente como modificador del valor de iteración empleado en la condición, aunque no es estrictamente necesario que en la condición figure el parámetro utilizado en la iteración, si por ejemplo se definen otras formas de salida dentro del contexto del bucle, y si se cumple la condición evaluada se ejecutarán las instrucciones dentro del contexto en caso de que el conjunto de condiciones del bucle se cumpla. Se ha considerado hacer que exista el `for-else`, que actuaría de tal manera que si no se cumple inicialmente la condición del primer bucle, comprobará la expresión del siguiente, y si su condición se cumple, estará evaluando repetidamente las instrucciones de este último (sin volver a comparar con las condiciones del bucle anterior al `else`). De esta manera, podemos hacer una estructura combinada de lo que en otros casos serían `if-for-else-for`, de forma directa con solo un `for-else`.

De forma análoga a los `else` de las expresiones condicionales, la representación para el `for-else` hará uso también del símbolo "."para indicar un `else` y sucesivamente los que pudiesen añadirse tras este, tantos como se deseen, considerando que el último else, al no tener condición de entrada como tal, se ejecutará una sola vez. En caso de que se aniden, hay que considerar que los cierres de contexto dependen del número de tabulaciones que hayan. A continuación se muestra un ejemplo de uso:

```
int a = 7
int b = 9
int suma = 0
int contador = 0

a > contador ?? contador++:
    suma = suma + a * contador + b
. b > contador ?? contador++:
    suma = suma + b*contador + a
.??:
    suma = a + b
.. Como 'a' es mayor que 'contador' inicialmente, solo se ejecutan instrucciones del primer bucle
```

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 2.

Listing 2: Declaracion de funciones

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

La definición de cualquier función sigue la estructura del listado 3. El cuerpo de la función debe comenzar por el caracter de cambio de contexto (caracter `0x09` o cuatro caracteres `0x20` seguidos).

Listing 3: Declaracion de funciones

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

El listado 4 ilustra la estructura general de llamada a una función.

Listing 4: 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.