Programación

Asignatura 240205 Programación.

Lectura[2]: **Tipos de datos**

Dr. José Ramón González de Mendívil
mendivil@unavarra.es
Departamento de Estadística, Informática y Matemáticas
Edificio Las Encinas
Universidad Pública de Navarra

Resumen

Dato es una palabra que en el contexto de la Informática[1] describe todo aquello con lo que puede operar un ordenador. Al nivel de los elementos con los que se construye un ordenador (circuitos electrónicos digitales, *hardware*) los datos están 'representados' como secuencias de dígitos binarios (bits, coloquialmente ceros y unos). Los lenguajes de alto nivel que empleamos en la actualidad para desarrollar programas (*software*), nos permiten pensar en los datos como lo hacemos habitualmente, de una manera más abstracta, ignorando por completo cómo se representan en las máquinas. A lo largo de los años 70 se fué definiendo de forma más precisa la noción de **Tipo de dato**. Presentamos en esta lectura los tipos de datos que vamos a emplear en la asignatura de Programación.

Índice

[1] Tratamiento automático de la información.

1. Tipos de variables y expresiones

En este curso de Programación vamos a desarrollar, en las clases de teoría, algoritmos que resuelven diferentes problemas, y en las clases de prácticas, codificamos y comprobamos dichos algoritmos utilizando el lenguaje de programación C. Todo lenguaje de programación tiene diferentes **tipos de datos** que tenemos que tener en cuenta.

Un **tipo de datos** define, por una parte, (1) un conjunto de posibles valores que forman el tipo; y (2) las posibles operaciones que podemos utilizar con los valores que forman el tipo.

En todo lenguaje de programación diferenciamos los tipos **simples** de los **estructurados**. A continuación presentaremos los tipos de datos que vamos a usar a lo largo del curso de Programación. Básicamente, los **tipos simples**: entero, intervalo de enteros, reales, booleanos, carácter, enumerado; y el **tipo estructurado**: tablas de los tipos anteriores.

Recordemos que para declarar una variable hay que indicar su 'nombre' y su 'tipo'. Lo que carácteriza a una variable es, efectivamente, poder cambiar su valor en cualquier momento. En cambio, una **constante** es un elemento inmutable. Declarar una costante es simplemente darle un 'nombre' e indicar su valor. En el lugar donde aparezca dicho 'nombre' se sustituye por su valor previamente declarado. Por convenio, los nombres de la constantes se escriben en mayúsculas. Ejemplo:

```
1: constante
2:     PI 3,1416
3:     N 1000
4: fconstante
```

2. Tipo entero

El conjunto de los números enteros, denotado $\mathbb{Z}$, contiene a los números naturales, el cero, y los números negativos. Todos los lenguajes de programación incluyen este tipo, y los ordenadores tienen formas adecuadas de representarlos en los circuitos electrónicos que utilizan.

Los 'literales' del tipo entero son los números enteros tal y como los escribimos. En todos los lenguajes de programación se escriben como lo hacemos habitualmente: 122, -33; y nunca usamos el punto o la coma decimal a la hora de escribir estos números. Así que para nosotros, los datos que forman el tipo entero son, simplemente, los números enteros. En un ordenador este tipo de datos tiene un **rango de representación**, es decir, en un ordenador real puede haber un límite en la capacidad de representación de los números enteros. De momento, no nos vamos a preocupar por esto[2].

Las expresiones (de tipo entero) que podemos formar con el tipo entero incluyen: variables declaradas como enteras, constantes establecidas como enteras, literales enteros (números enteros), uso de paréntesis, y las operaciones siguientes:

+	suma
-	resta (o cambio de signo)
*	multiplicación
div	cociente de la división entera
mod	resto de la división entera
max	máximo de dos enteros
min	mínimo de dos enteros

[2]Tienes que ocuparte de estos detalles cuando codificas un programa en un lenguaje de programación. En C tienes distintos tipos enteros que se diferencian por el rango de números con los que se puede operar.

Considera la declaración siguiente,

```
1: var
2:     a, b, x, y: entero
3: fvar
```

En la formación de expresiones del tipo entero hay que tener en cuenta las **reglas de prioridad** de las operaciones en el caso de no usar paréntesis. La tabla anterior indica que la suma es la operación menos prioritaria y min la más prioritaria. Además, cuando se escribe una expresión, está debe estar **sintácticamente bien formada**[3]. Ejemplos de expresiones de tipo entero:

$a \bmod b + 3 * x$, es equivalente a escribir $(a \bmod b) + (3 * x)$
$3 + x * y$, equivalente a escribir $3 + (x * y)$
$\max(x + 6, y) + a \operatorname{div} b$, equivalente a escribir $\max(x + 6, y) + (a \operatorname{div} b)$

Las expresiones anteriores están bien formadas, pero el valor que pueden tomar dependen del **estado de asignación** de las variables correspondientes. Consideremos el estado σ es $\{a = 5 \wedge b = 3 \wedge x = 2 \wedge y = 7\}$, la siguiente tabla muestra el resultado de ejecutar diferentes sentencias de asignación a partir de dicho estado (completa la tabla):

sentencia	calcula el nuevo estado a partir de σ
$x := (a \bmod b) + (3 * x)$	–
$y := 3 + (x * y)$	–
$x := \max(x + 6, a \bmod b) + (a \operatorname{div} (y - x))$	–

Las expresiones de tipo entero, además de tener que ser sintácticamente correctas, tienen que poder evaluarse sin producir ningún error. Recordad que la relación entre el cociente q (A div B) y el resto r (A mod B) de la división de dos números enteros A y B, siendo $B \neq 0$, tiene que cumplir lo siguiente:

$$A = q * B + r \wedge 0 \leq r < |B| \tag{1}$$

Si escribimos una instrucción de asignación como $x := x$ div b y la ejecutamos en un estado donde el valor de b es 0 tendremos un error en la ejecución. La elaboración de **algoritmos correctos** trata de que los programas que desarrollemos no contengan errores cuando se ejecuten, y realicen el propósito para el cuál se han diseñado. Para cada expresión entera podemos definir un predicado[4] *que indica las condiciones para que la expresión se evalúe sin error*. Usamos el término def() para éste propósito. Ejemplos:

def$((a \bmod b) + (3 * x))$ es equivalente a la condición $b \neq 0$
def$(\max(x + 6, a \bmod b) + (a \operatorname{div} (y - x)))$ es equivalente a las condiciones $y \neq x \wedge b \neq 0$

Hemos incluido en el tipo entero las operaciones max() y min(). Algunos lenguajes de programación las incluyen por defecto y en otros no. En estos últimos, ampliar el repertorio de instrucciones de un determinado tipo significa ir construyendo funciones apropiadas e incluirlas en alguna **librería** para utilizarla posteriormente. En cualquier caso, plantear el diseño de una función para el max() o min() de números enteros no resulta muy complicado.

[3] No damos la sintáxis completa de las expresiones aritméticas. Los estudiantes deben saber diferenciar una expresión aritmética bien escrita de otra que no lo está.

[4] Los predicados los trataremos de forma más precisa un poco más adelante.

Nota: Me parece importante incluir todas las posibles composiciones de sentencias y su funcionamiento operacional. En este caso presentamos la composición alternativa **si**...**fsi**. En la declaración de las funciones a, b: entero son **parámetros formales de entrada** y res: entero es el **parámetro formal de salida**. Cuando se usa una función, como en los ejemplos de la tabla anterior, las expresiones dadas sustituyen a los parámetros formales y se dice que son los **parámetros reales** sobre los que actúa la función.

```
1:  funcion max (a, b: entero) dev res: entero
2:      {cierto}
3:      si
4:          [] a > b ⟶
5:                          res := a
6:          [] a ≤ b ⟶
7:                          res := b
8:      fsi;
9:      {res = max(a, b)}
10:     dev (res)
11: ffuncion
```

```
1:  funcion min (a, b: entero) dev res: entero
2:      {cierto}
3:      si
4:          [] a < b ⟶
5:                          res := a
6:          [] a ≥ b ⟶
7:                          res := b
8:      fsi;
9:      {res = min(a, b)}
10:     dev (res)
11: ffuncion
```

3. Representación de los estados de ejecución de un programa

3.1. Nociones elementales de conjuntos

Cuando queremos tratar a un 'grupo de objetos', por las razones que sean, como una entidad, los agrupamos formando **conjuntos**. Podemos definir el conjunto por extensión, indicando de forma explícita los elementos que forman parte de él. Agrupamos los elementos entre llaves, $\{id1, id2, id3, ..., idN\}$ donde id_k representa el símbolo elemento k-ésimo en el conjuno. Una vez formado el conjunto le podemos dar un nombre para trabajar con él como un todo: Sea A el conjunto $\{id1, id2, id3, ..., idN\}$, o simplemente, $A \stackrel{\text{def}}{=} \{id1, id2, id3, ..., idN\}$[5].

Algunos conjuntos por ser más 'universales' tienen un nombre predeterminado como por ejemplo $\mathbb{N}$ para el conjunto de los números naturales, o $\mathbb{Z}$ para el conjunto de los números enteros. Fundamental a la noción de conjunto es la **relación de pertenencia**: $3 \in \mathbb{N}$ se lee 'tres pertenece a los números naturales'. La relación contraria de 'no pertenencia', se escribe con el símbolo $\notin$; ejemplo, $-3 \notin \mathbb{N}$.

Sean dos conjuntos A y B. Se dice que son iguales cuando todo elemento de A también lo es de B y viceversa, y se escribe $A = B$. En caso contrario, cuando son distintos se escribe $A \neq B$.

[5]El símbolo $\stackrel{\text{def}}{=}$ se lee 'es igual por definición a'.

Si cada elemento de A es elemento también de B entonces se dice que A está incluido en B, $A \subseteq B$. En ese caso decimos que A es un **subconjunto** de B.

Los conjuntos se pueden unir, de forma que si A y B son dos conjuntos su unión se representa como $A \cup B$. Este conjunto contiene los elementos que pertenecen a ambos A y B (no se repiten). La intersección $A \cap B$ es el conjunto que contiene a los elementos comunes a ambos. Si no hay elementos comunes entonces el resultado es el **conjunto vacío**, denotado como $\emptyset$. El conjunto vacío es en toda regla un conjunto que tiene el siguiente comportamiento: $A \cup \emptyset = A$ y $A \cap \emptyset = \emptyset$ para todo conjunto A.

Para definir subconjuntos es conveniente utilizar las propiedades que tienen los elementos que van a formar parte del subconjunto. Ejemplos: sea x un número entero,

- El subconjunto de 'los enteros mayores estrictos que 23': $\{x \mid x > 23\}$.
- El subconjunto de 'enteros entre -15 y 28': $\{x \mid -15 \leq x \leq 28\}$.
- El subconjuto de 'enteros positivos que son divisibles por 3': $\{x \mid x > 0 \wedge x \bmod 3 = 0\}$.

Para que un número entero pertenezca a uno de los subconjuntos dados anteriormente se debe cumplir la propiedad dada sobre dicho número, por ejemplo, $18 \notin \{x \mid x > 23\}$ ya que $18 > 23$ es falso, mientras que $42 \in \{x \mid x > 23\}$ puesto que $42 > 23$ es cierto.

Entre las construcciones más notables que se pueden realizar con conjuntos es la formación de parejas (en general tuplas) ordenadas entre los mismos. Sean dos conjuntos A y B, entonces el conjunto **producto cartesiano** de ambos, denotado $A \times B$, es el conjunto definido como $A \times B \stackrel{\text{def}}{=} \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$.

3.2. Notación para representar Estados de ejecución

Consideremos, por comodidad, la declaración de variables enteras

```
1: var
2:     a, b, c : entero
3: fvar
```

Un programa que utiliza estas variables en su ejecución puede modificar el valor de las mismas. El **estado de ejecución** en cada momento está determinado por los valores concretos que toman dichas variables en ese momento. Podemos, representar el estado de ejecución en un determinado momento de la ejecución de un programa como una tupla, por ejemplo, $\sigma_1 = \langle -3, -1, 4\rangle$ representa el estado en el instante 1, o $\sigma_2 = \langle -4, -1, 8\rangle$ representa el estado en el instante 2. Para que esto tenga sentido debemos indicar que la primera coordenada es para la variable a, la segunda para b y la tercera para c. Así, también podemos indicar que $\sigma_1(a)$ es el valor de la variable a en el estado σ_1. Esta última notación es más cómoda cuando tenemos muchas variables de diferentes tipos[6]. La declaración de variables dada nos indica además, que el conjuntos de todos los estados posibles son todas las tuplas que podríamos formar con los valores de las variables. Si convenimos en denotar por Γ este conjunto entonces, para la declaración dada, $\Gamma \stackrel{\text{def}}{=} \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

Como el conjunto de todos los estados de ejecución posibles Γ es un 'conjunto' podemos definir diferentes subconjuntos dando las propiedades que queremos que cumplan sus estados. Las

[6]Entendemos también el estado de ejecución (o de asignación) como una función que asigna a cada 'nombre' de variable un valor de su tipo $\sigma : ID \to D$

propiedades se representan mediante **predicados**. En general, dada una declaración de variables y un predicado P el subconjunto de estados determinado por dicho predicado lo definimos como:

$$\{P\} \stackrel{\text{def}}{=} \{\sigma \in \Gamma | \text{ se cumple } P \text{ sobre } \sigma\} \tag{2}$$

Ejemplos. Calcula los subconjuntos de estados siguientes:

- $\{a = 2 \wedge b = -4 \wedge c = 0\} = \{\langle 2, -4, 0\rangle\}$
- $\{1 \leq a \leq 2\} = \{\langle 1, b, c\rangle, \langle 2, b, c\rangle \,|\, b, c \in \mathbb{Z}\} = \{\langle 1, -, -\rangle, \langle 2, -, -\rangle\}$[7]
- $\{a \geq b \geq c\} = \{.....\}$
- $\{a \geq b\} = \{.....\}$
- $\{a \bmod 2 = 0 \wedge c = 0\} = \{.....\}$
- $\{2 \leq a \leq 1\} = \{.....\}$
- Dado un número A, $\{s = A \wedge s > 0\} = \{.....\}$

Observa que $\{a \geq b \geq c\} \subset \{a \geq b\}$. Observa también el primer ejemplo. Sólo contiene un estado, justamente el único que hace cierto el predicado $a = 2 \wedge b = -4 \wedge c = 0$. Por ese motivo, también representamos un estado de ejecución de esa manera como lo hicimos en los ejemplos de la Lectura[1]. A la vista de lo indicado en esta sección su interpretación debe quedar clara al estudiante.

4. Tipo intervalo de enteros

En algunos programas interesa acotar los números enteros a utilizar por una determinada variable entre dos límites. Ejemplos,

```
1: constante
2:     A 0
3:     B 999                          ▷ límites de las posiciones
4: fconstante
5: var
6:     k: 1 .. 10                     ▷ k puede tomar los valores entre 1 y 10
7:     pos: A .. B                    ▷ pos puede tomar los valores entre A y B, A ≤ B
8: fvar
```

La variable k puede 'recorrer' exáctamente 10 números ente 1 y 10, ambos incluidos; la variable *pos* puede recorrer 1000 números entre 0 y 999 (declarados previamente como constantes). Los intervalos en ambos casos están bien definidos: el número a la izquierda de '..' es menor o igual que el que está a la derecha de '..'. En caso contrario el intervalo estaría vacío.

Un lenguaje de programación que permite este tipo de declaración es Pascal. También se le suele denominar 'tipo subrango de enteros'. Este tipo de declaración obliga, en la ejecución de las instrucciones de asignación que contienen este tipo de variables, a comprobar que los valores de las variables no se salen de los rangos establecidos. Ejemplo: según la declaración anterior, la ejecución de la instrucción de asignación $k := 12$ produciría un error al salirse la variable de su rango. El lenguaje C no tiene este tipo de declaraciones[8].

[7] $\langle 1, -, -\rangle$, el guión $-$ indica que la variable puede tomar cualquier valor de su tipo.

[8] Aunque el lenguaje no lo tenga, si esperas confinar a una variable dentro de un rango de valores enteros para que tu programa funcione bien es aconsejable poner algún comentario.

Las variables del tipo intervalo de enteros se consideran variables enteras a todos los efectos excepto por el hecho de que sus valores no pueden salirse del rango declarado.

Nota: Las operaciones posibles en la variable *pos* anterior nunca se salen del rango si utilizas la aritmética modular. Ejemplo, $pos := (pos + 2903) \bmod 1000$, ya que los restos de dividir por 1000 son justamente los números [0 .. 999].

Nota: Suponer que A y B son dos literales de números enteros o bien constantes enteras. Otras formas de declarar intervalos de enteros que posiblemente utilicemos serán:

- $[A \,..\, B]$ representa el subconjunto $\{\gamma \in \mathbb{Z} \mid A \leq \gamma \leq B\}$
- $[A \,..\, B)$ representa el subconjunto $\{\gamma \in \mathbb{Z} \mid A \leq \gamma < B\}$
- $(A \,..\, B]$ representa el subconjunto $\{\gamma \in \mathbb{Z} \mid A < \gamma \leq B\}$
- $(A \,..\, B)$ representa el subconjunto $\{\gamma \in \mathbb{Z} \mid A < \gamma < B\}$

5. Tipo booleano

Una **proposición** es una sentencia del lenguaje que tiene la característica de ser cierta o falsa, pero sólo una de las dos cosas es posible por el *principio de exclusión*[9]. Los términos *cierto* ('verdadero') o *falso* son los llamados *valores veritativos* de las proposiciones y se pueden denotar también con las letras V y F. En algunos textos también se emplea 1 y 0 como sinónimos de cierto y falso[10].

Ejemplos: '5 es un número primo'; '7 mas 3 es igual a 8'; '-4 no pertenece a los naturales'; 'la tierra es redonda' (coloquial).

La primera proposición es cierta, la segunda falsa, la tercera es cierta, y la cuarta...depende, hay gente que sigue pensando que la tierra es plana. La Lógica matemática no pretende conocer la 'verdad' (en términos filosóficos de 'realidad') de las proposiciones, más bien, obtener deducciones lógicas en concordancia con las operaciones y valores veritativos de las proposiciones que se formulan.

Las proposiciones, como sentencias del lenguaje, pueden combinarse gramaticalmente para formar nuevas proposiciones: la conjunción lógica, la negación o la disyunción lógica son formas que utilizamos habitualmente para ese proposito. Si p y q representan **valores veritativos**, podemos formar la siguiente tabla para el cálculo con las operaciones lógicas más comunes:

		y lógico	o lógico	negación	condicional	bicondicional
p	q	$p \wedge q$	$p \vee q$	$\neg p$	$p \Rightarrow q$	$p \equiv q$
V	V	V	V	F	V	V
V	F	F	V	F	F	F
F	V	F	V	V	V	F
F	F	F	F	V	V	V

[9] Esto no significa que no haya otro tipo de lógicas matemáticas que han sido desarrolladas posteriormente, donde este principo de exclusión está relajado. De notable interés ha sido el desarrollo de la lógica multivaluada y de la lógica difusa (fuzzy).

[10] Los problemas con estos conceptos tan simples aparecen cuando algunos programadores interpretan también 1 y 0 como números y cómo valores de verdad. Bueno eso aparece en el lenguaje C, las cosas no siempre son perfectas.

Debemos pensar que V y F son valores del conjunto $\mathbb{B} \stackrel{\text{def}}{=} \{V, F\}$, y que $\wedge$, $\vee$, $\neg$, $\Rightarrow$, y $\equiv$ son operaciones que se pueden aplicar a dichos valores. Normalmente el orden de prioridad es el siguiente (de mayor a menor): $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$. Así, la expresión $p \Rightarrow q \equiv \neg p \vee q$ es equivalente a escribir $(p \Rightarrow q) \equiv ((\neg p) \vee q)$.

De la tabla anterior se pueden deducir ciertas propiedades que son importantes:

- La operación de conjunción $\wedge$ es una operación asociativa, conmutativa, con elemento neutro V.
- La operación de disyunción $\vee$ es asociativa, conmutativa, con elemento neutro F.
- Para cualesquiera valores veritativos p, q y r las siguientes expresiones son siempre ciertas:
  - La conjunción y la disyuncción son ambas distributivas entre sí:
    $(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$
    $(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r)$
  - La doble operación de negación: $\neg(\neg p) \equiv p$
  - La operación condicional: $p \Rightarrow q \equiv \neg p \vee q$
  - La operación bicondicional[11]: $(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \equiv q$

5.1. Variables y expresiones booleanas

Si con los números enteros podemos declarar variables enteras para construir expresiones enteras, entonces, con los valores de veritativos también podemos declarar variables booleanas[12] para construir expresiones booleanas. Una variable booleana sólo puede tomar los valores cierto o falso (escribimos tambien V o F siempre que no haya confusión) y ningún otro.

Entonces para formar expresiones booleanas (simples) podemos emplear: los literales del tipo booleano **cierto** o **falso**, constantes declaradas con dichos valores, variables del tipo booleano, paréntesis y las operaciones $\wedge$, $\vee$, o $\neg$.

Según la tabla anterior y las propiedades que hemos visto, es suficiente con esas tres operaciones para definir las otras operaciones booleanas.

Considera la declaración siguiente, y sea σ el estado $\{a = V \wedge b = F \wedge c = V\}$.

```
1: var
2:     a, b, c: booelano
3: fvar
```

asignación	calcula el siguiente estado a partir de σ
$a := \neg(a \wedge b)$	–
$b := a \wedge \neg b \vee c$	–
$c := a \wedge b \wedge c$	–
$c := a \equiv b$	–

[11] También la denomino al hablar como 'equivalencia' o 'igualdad lógica'. Observa que no he puesto el paréntesis en $(p \equiv q)$. Los estudiantes pueden comprobar que se cumple la expresión dada.

[12] Se denominan así en homenage a George Boole (1815–1865) creador del cálculo lógico.

Ejercicio 1 *En [1]. Hay que construir un circuito eléctrico para el mando de un elevador; suponemos que el número de pisos es dos, para simplificar. El circuito debe contener dos contactos que se manipulan oprimiendo botones instalados en la cabina del elevador (botón de descenso) y en el primer piso, junto a la puerta del elevador (botón de llamada); los contactos adicionales están relacionados con las puertas del elevador en el primer piso y en el segundo piso, con la puerta interior de la cabina, así como con el suelo del elevador sobre el cual ejerce presión el pasajero que se encuentra en la cabina. El circuito eléctrico que permite manejar el descenso del elevador, debe conectarse sólo si la cabina se encuentra en el segundo piso y si, además, se cumplen las condiciones siguientes: 1) Están cerradas ambas puertas del elevador, y la puerta de la cabina, el pasajero se encuentra en la cabina y oprime el botón de descenso o, 2) están cerradas ambas puertas del elevador (mientras que la puerta de la cabina está cerrada o abierta); en la cabina no hay nadie; una persona oprime el botón en el primer piso de llamada.*
Escribe la expresión booleana que se requiere para construir el circuito siguiendo la lógica anterior.

El ejercicio anterior es un ejemplo bastante sencillo de la utilidad de las expresiones booleanas en la toma de decisiones. Este aspecto es habitual en la ejecución de los programas. El desarrollo de la Arquitectura de Ordenadores se basa en los denominados circuitos lógicos combinacionales cuya formación y composición surge de las operaciones de la lógica matemática.

Nota: Como curiosidad, la toma de decisiones con expresiones booleanas ha interesado a los investigadores durante batante tiempo ver por ejemplo las explicaciones en [3] sobre el trabajo de Huang [4].

5.2. Operaciones de relación y expresiones booleanas

Cuando escribimos '3 = 3', y leemos 'tres es igual a tres', tenemos bastante claro que es cierto. Ahora bien cuando escribimos '$x = 3$' y decimos 'x es igual a tres', al menos yo no tengo tan claro que sea cierto. Tienes que poner algo más de contexto a la frase. Sea x una variable de tipo entero, $x = 3$ es una expresión de relación 'igualdad' que tomará valor cierto cuando el valor de la variable x sea 3. En otras palabras, la expresión $x = 3$ es una expresión booleana que se somete a evaluación en un determinado estado de asignación. Por este motivo, la asignación $b := (x = 3)$ es una asignación correcta si b se declara como booleano.

Queremos decir con lo anterior, que podemos construir expresiones booleanas más complicadas que las ofrecidas al comienzo de esta sección.

Para formar expresiones booleanas podemos emplear:

- los literales del tipo booleano **cierto** o **falso**, constantes declaradas con dichos valores, variables del tipo booleano, paréntesis y las operaciones $\wedge$, $\vee$, o $\neg$.
- Expresiones de tipo entero, o expresiones de tipo real, que se relacionan entre ellas con los operadores de relación siguientes: $=$, $\neq$, $>$, $\geq$, $<$, $\leq$.

En cuanto a la prioridad, las operaciones de relación aritméticas tienen mayor prioridad que las operaciones lógicas[13] y menor prioridad que las operaciones aritméticas.

Nota: En los predicados permitimos escribir $0 \leq x \leq 10$, pero si debes programar esa expresión como una expresión boolena deberás escribir $0 \leq x \wedge x \leq 10$.

[13]Esta es también una buena razón para no usar = entre expresiones booleanas y emplear $\equiv$. Esta distinción simplifica la interpretación de los predicados.

Finalmente, indicar que las **expresiones booleanas** son también **predicados** pero los predicados son más generales ya que vamos a poder incluir en ellos otras expresiones como veremos en una próxima Lectura.

6. Otros tipos básicos: real, caracter, enumerado

En los lenguajes de programación encontramos también el tipo básico 'real', para indicar que una variable puede contener un número racional (con decimales). Escribimos los literales del tipo real como lo hacemos habitualmente con la notación científica, 12.34, $3.0E-10$ (3×10^{-3}). El rango del tipo 'real' depende del ordenador que utilices pero el conjunto de números reales que puedes emplear siempre será un subconjunto de los números racionales. Las operaciones con este tipo son: +, -, *, /. La división / y la multiplicación entre dos reales puede dar un valor aproximado al resultado exacto. Esta es una diferencia notable con respecto al tipo entero, donde los resultados de las operaciones son exactas. La representación en la memoria de un ordenador de un número real puede utilizar diferente rango y distinta precisión. El estudiante puede consultar el estándar IEEE754 para ver cómo se codifican los números reales. En las expresiones donde uses variables reales también puedes utilizar variables enteras (al revés no!). Ejemplo: r: real, x: entero. $r := r/2 + x$ es una asignación correcta, pero $x := r/2 + x$ no es correcta, no casa el tipo de la expresión con el tipo de la variable asignada. Cada lenguaje de programación utiliza distintas reglas y hay que conocerlas para no comenter errores cuando codificas los algoritmos. Las comparaciones entre variables reales mediante $=$, $\neq$, $>$, etc..., también son permitidas.

El tipo 'caracter' hace referencia a los caracteres que escribimos con el teclado (y algunos otros más de control de un texto: salto de línea, fin de texto, etc.). Una variable p: caracter, puede contener un símbolo que hace referencia a un elemento del teclado. Los literales del tipo carácter los representamos entre comillas, por ejemplo, 'a', 'n', '2', representan los carácteres asociados a dichas teclas. La codificación que se emplea (la más básica) es el **código ASCII**. Cada carácter tiene un número asociado dada por la codificación ASCII, por ejemplo, 'a' es el número decimal 97 y '2' es el número decimal 50. No se debe confundir '2' con el número entero 2. Son dos cosas diferentes. El lenguaje C te permite usar los carácteres como números enteros, por ejemplo, '2' + 2 es el entero 52. Podemos utilizar las comparaciones entre variables del tipo carácter, por ejemplo, *continuar* = 's' será cierto si la variable de tipo carácter *continuar* contiene el valor 's'.

En algunos programas se requiere por claridad que el programador cree sus propios valores. En este caso, el programador dá la lista de valores que puede contener la variable. Por ejemplo, la siguiente declaración es de un tipo 'enumerado', $estado\_calefacion$: (encendido, apagado, error). Entre paréntesis indicamos los valores. Podemos hacer una instrucción de asignación como, $estado\_calefacion$:= encendido, o comparar, $estado\_calefacion$ = apagado. No entramos aquí en más detalles. Los estudiantes deben leer la primera sesión del guión de prácticas.

Ejercicio 2 *a) Consulta la representación de números enteros en signo-magnitud, complemento a uno, complemento a dos. b) Consulta el estándar IEEE754. c) Consulta el código ASCII. d) Lea la primera sesión del guión de prácticas en miaulario.*

7. Tipo estructurado: tablas

Con declaraciones de variables de tipos simples no podemos realizar de manera sencilla programas que traten con estructuras más complejas. No resulta sencillo hacer programas que realicen cálculos con matrices o vectores, o por ejemplo, programas que imiten juegos como el

ajedrez, una simple 'guerra de barcos', o programas que busquen en un determinado texto determinadas palabras. Las **tablas** ('arrays'), desde un punto de vista formal, son funciones sobre un subconjunto finito de enteros, al que denominamos **índice**[14] de la tabla. Para declarar una tabla debemos indicar el índice y el tipo de los elementos.

Ejemplo: t: **tabla** [1 .. 10] **de** entero

La declaración es correcta si el índice de la tabla no está vacío[15]. La variable t, es una tabla que tiene 10 elementos cuyos índices van desde el 1 hasta el 10; y cada elemento de la tabla contiene un número entero (su **tipo base**). Si quieremos acceder al valor en un determinado índice, por ejemplo, el 3, debemos escribir $t[3]$. El acceso a los elementos de la tabla es un **acceso directo**: conocida la posición se puede acceder inmediatamente al elemento y su valor. Las tablas las podemos emplear en sentencias de asignación del tipo base de la tabla. Considera el siguiente fragmento de programa,

```
1:  constante
2:      N 10                                  ▷ N ≥ 1, para que la tabla esté definida
3:  fconstante
4:  var
5:      i, j: entero
6:      v: tabla [0 .. N − 1] de entero       ▷ Posiciones desde 0 hasta N − 1
7:  fvar
8:  i:= 0;
9:  j:= N;
10: mientras i < N hacer
11:     v[i]:= 2 * i + j;                     ▷ inicializa los valores de cada posición de la tabla
12:     i:= i + 1
13: fmientras;
14: ....
15: ....
```

Cuando la ejecución del fragmento anterior alcanza la línea 14, en cada posición p desde 0 hasta $N-1$, $v[p]$ vale $2*p+N$ según el programa anterior. Cuando hemos diseñado el programa debemos asegurarnos que la asignación $v[i]:= 2 * i + j$ está bien definida, es decir,

def($v[i]:= 2 * i + j$) es equivalente a $0 \leq i \leq N-1$

La variable i en esa instrucción de asignación tiene que 'apuntar' a una posición válida de la tabla, en el caso que nos ocupa sus valores tienen que estar comprendidos entre 0 y $N-1$. El programa anterior lo cumple ya que la variable i, comienza en 0 y termina en N, pero dentro del **mientras** no se accede a $v[N]$, ya que se entra si $i < N$. El acceso a un índice fuera de la declaración es un error (que puede ser grave si no se detecta). En general si la declaración de la tabla es t: **tabla** [L .. U] **de** entero, para dos constantes dadas L y U, cualquier acceso a un elemento de la tabla mediante una expresión Exp, $t[Exp]$, debe cumplir la condición de 'bien definido', def($t[Exp]$) es equivalente a def(Exp) $\wedge\ L \leq Exp \leq U$.

Podemos declarar tablas con varios índices (tablas multidimensionales). Ejemplos (sea $N > 0$ una constante dada):

[14] También le podemos llamar conjunto de posiciones de la tabla.

[15] Algunas especificaciones de problemas con tablas admiten que el índice de la tabla esté vacio. Su utilidad depende del problema a tratar y de su generalizazión mediane acciones o funciones.

- *mat*1, *mat*2: **tabla** [1 .. N][1 .. N] **de** real
- *horario*: **tabla** [8 .. 14][1 .. 5] **de** asignatura
- *tabajedrez*: **tabla** [1 .. 8][1 .. 8] **de** piezas

En el primer ejemplo hemos declarado dos matrices cuadradas de N-filas por N-columnnas. Cuando queremos acceder a la fila 2 columna 3 de la matriz *mat*1, lo escribimos como $mat1[2, 3]$, o como $mat1[2][3]$. En el segundo ejemplo hemos puesto como tipo base el tipo 'asignatura' que se debe declara como un tipo enumerado. Lo mismo sucede en el tercer ejemplo.

Como ejemplo de uso de las tablas, en el siguiente fragmeno de programa mostramos cómo calcular la transpuesta de una matriz:

```
1: constante
2:     N 3                                                    ▷ N ≥ 1
3: fconstante
4: tipo
5:     matriz :: tabla [1 .. N][1 .. N] de real      ▷ define el tipo 'matriz' por claridad
6: ftipo
7: var
8:     i, j: 1 .. N
9:     mat, trans: matriz
10: fvar
11: ....
12: ....                  ▷ suponemos que la matriz mat ya contiene valores en sus elementos
13: i:= 1;
14: mientras i ≤ N hacer
15:     j:= 1;
16:     mientras j ≤ N hacer
17:         trans[j, i] := mat[i, j];
18:         j:= j + 1
19:     fmientras;
20:     i:= i + 1
21: fmientras;
22: ....                                          ▷ trans es la transpuesta de mat
```

Cabeceras de programas. En las cabeceras de los programas, como en el fragmento anterior, se indican primero las constantes, luego se declaran tipos, y posteriormente la declaración de variables. También se suelen declarar antes de las variables los prototipos de las funciones y acciones que se van a emplear. La razón de hacerlo así es por claridad a la hora de leer el texto del programa. La razón de declarar las funciones antes de las variables que se usan en el programa es para detectar que no se usan en las funciones variables que no han sido declaradas localmente en las propias funciones, evitando así el uso de variables globales.

Nota: En el programa anterior, que calcula la transpuesta de una matriz, las estructuras

i:= 1; **mientras** $i \leq N$ **hacer**....i:= $i + 1$ **fmientras**
j:= 1; **mientras** $j \leq N$ **hacer**....j:= $j + 1$ **fmientras**

se anidan; la segunda dentro de la primera. Si ejecutas la primera observa que dentro del bucle los valores que toman la variable i son exáctamente 1, 2, ..., N. Si ejecutas las segunda, la variable j, dentro de su bucle, toma también los valores 1, 2, ..., N. Para mejorar la claridad, y por ser muy común en programación también se escriben como en los siguientes ejemplos. En el primer ejemplo, se usa la misma forma

que en el lenguaje C. En el segundo ejemplo se usa la misma forma de Pascal.

```
1: constante
2:     N 3                                                          ▷ N ≥ 1
3: fconstante
4: tipo
5:     matriz :: tabla [1 .. N][1 .. N] de real                     ▷ define el tipo 'matriz' por claridad
6: ftipo
7: var
8:     i, j: 1 .. N
9:     mat, trans: matriz
10: fvar
11: ....
12: ....                                  ▷ suponemos que la matriz mat ya contiene valores en sus elementos
13: para (i:= 1, i ≤ N, i:= i + 1) hacer
14:     para (j:= 1, j ≤ N, j:= j + 1) hacer
15:         trans[j, i] := mat[i, j];
16:     fpara
17: fpara
18: ....                                                  ▷ trans es la transpuesta de mat
```

```
1: constante
2:     N 3                                                          ▷ N ≥ 1
3: fconstante
4: tipo
5:     matriz :: tabla [1 .. N][1 .. N] de real                     ▷ define el tipo 'matriz' por claridad
6: ftipo
7: var
8:     i, j: 1 .. N
9:     mat, trans: matriz
10: fvar
11: ....
12: ....                                  ▷ suponemos que la matriz mat ya contiene valores en sus elementos
13: para i:= 1 hasta N hacer
14:     para j:= 1 hasta N hacer
15:         trans[j, i] := mat[i, j];
16:     fpara
17: fpara
18: ....                                                  ▷ trans es la transpuesta de mat
```

El primer uso del **para**...**hacer** es una reescritura del **mientras** y tienen toda su capacidad. El segundo uso del **para**...**hasta**...**hacer**, es más restrictivo ya que sólo sirve para ir llevando a la variable desde el primer valor al último de uno en uno.

Indicaciones para los estudiantes: A lo largo de estas dos primeras lecturas hemos visto las sentencias que empleamos en la construcción de programas: asignación, composición secuencial (el ';'), composición alternativa **si**...**fsi**, composición iterativa **mientras**...**fmientras**. La declaración de variables y sus tipos: tipos básicos (entero, booleano, carácter, enumerado), y el tipo estructurado, **tabla**...**de** ... Llegados a este punto, los estudiantes deben saber ejecutar las sentencias de un programa sencillo dado y entender el concepto de estado de ejecución de un programa (descrito por los valores que toman las variables en un determinado momento de la ejecución). También deben comprender cómo se forman conjuntos de estados a partir de las propiedades, indicadas éstas mediante predicados, que cumplen las variables. Los predicados los tratarémos más detalladamente en la próxima Lectura[3].

Nota personal: ¿y a partir de aquí...? En un gran número de textos de informática se presentan soluciones correctas a diferentes problemas típicos de la programación basandose en ideas obtenidas de los trabajos previos realizados por otros 'grandes' programadores. Estás soluciones se explican, pero no se ofrece un método sistemático para su obtención. La programación siguiendo esta forma explicativa de programas correctos conduce a una programación por imitación, ya que los estudiantes intentarán 'ajustar' soluciones que ya conocen a nuevos problemas, entrando en ciclos de prueba y error (que muchas veces son frustrantes). En esta asignatura de Programación el camino es distinto. Se ofrece un método sistemático para obtener programas correctos para los problemas típicos de programación. El camino es un poco más arduo porque requiere del uso de la lógica matemática pero, la recompensa se encuentra en una comprensión más profunda de los 'fundamentos de la programación' que son la base para la 'computación', entendida ésta como 'ciencia de la computación'.

8. Ejercicios

Los programas que realizamos tienen que ser correctos: comenzando en cualquier estado que cumple unas determinadas propiedades iniciales, al ejecutar las sentencias del programa desde ese estado: (i) el programa debe terminar su ejecución, es decir, alcanza un estado de ejecución final; y (ii) el estado final alcanzado, debe ser un estado tal que cumpla con aquellas propiedades para las que se ha diseñado el programa.

Ejercicio 3 *Algunos programas por ser bastante simples, su corrección se puede demostrar ejecutando las propias sentencias que lo forman. Este es el caso del siguiente ejercicio donde se pide 'intercambiar' los valores entre dos variables. Comprueba que terminan y que realizan su propósito.*

1. *Intercambio con una sentencia de asignación múltiple.*

```
1: var
2:     x, y: entero
3: fvar
4: {x = A ∧ y = B}          ▷ estado inicial, x e y toman valores A y B respectivamente
5: ⟨x, y⟩ := ⟨y, x⟩
6: {x = B ∧ y = A}          ▷ estado final, x e y han intercambiado sus valores iniciales
```

2. *Intercambio mediante cálculo de la diferencia.*

```
1: var
2:     x, y: entero
3: fvar
4: {x = A ∧ y = B}          ▷ estado inicial, x e y toman valores A y B respectivamente
5: x := x − y;
6: y := x + y;
7: x := y − x
8: {x = B ∧ y = A}          ▷ estado final, x e y han intercambiado sus valores iniciales
```

3. *Intercambio mediante una variable auxiliar.*

```
1: var
2:     x, y: entero
3: fvar
4: {x = A ∧ y = B}          ▷ estado inicial, x e y toman valores A y B respectivamente
5: var aux: entero fvar                    ▷ declaración de una variable auxiliar
```

```
6: aux := x;
7: x := y;
8: y := aux
9: {x = B ∧ y = A}          ▷ estado final, x e y han intercambiado sus valores iniciales
```

Ejercicio 4 *Consideramos el mismo problema del intercambio, pero en esta caso para intercambiar los valores entre dos posiciones dadas de una tabla de enteros.*

```
1: constante
2:     N 10
3: fconstante
4: var
5:     i, j: entero
6:     t : tabla [1..N] de entero
7: fvar
8: {t[i] = A ∧ t[j] = B ∧ 1 ≤ i ≤ N ∧ 1 ≤ j ≤ N}
9:                          ▷ estado inicial, t[i] e t[j] toman valores A y B respectivamente
10: .....
11: .....
12: {t[i] = B ∧ t[j] = A ∧ 1 ≤ i ≤ N ∧ 1 ≤ j ≤ N}
13:                          ▷ estado final, t[i] e t[j] han intercambiado sus valores iniciales
```

La cuestión es ¿cuál de las tres soluciones anteriores no es correcta?. Basta con que para un estado inicial no se cumplan los requisitos finales para que la solución no sea correcta. Para responder a la cuestión adapta primero las sentencias para que funcionen con la tabla dada.

Ejercicio 5 *Ejecuta las funciones dadas para el cáculo del máximo y del mínimo dadas en la sección 2. Comprueba que son correctas. Debes comprobar todos los casos posibles de estados iniciales.*

Ejercicio 6 *Intercambio de variables booleanas.*

1. *Intercambio con una sentencia de asignación múltiple.*

```
1: var
2:     a, b: boleano
3: fvar
4: {(a ≡ A) ∧ (b ≡ B)}      ▷ estado inicial, a e b toman valores A y B respectivamente
5: ⟨a, b⟩ := ⟨b, a⟩
6: {(a ≡ B) ∧ (b ≡ A)}      ▷ estado final, a e b han intercambiado sus valores iniciales
```

2. *Intercambio mediante operaciones de igualdad lógica.*

```
1: var
2:     a, b: boleano
3: fvar
4: {(a ≡ A) ∧ (b ≡ B)}      ▷ estado inicial, a e b toman valores A y B respectivamente
5: a := a ≡ b;
6: b := a ≡ b;
7: a := a ≡ b
8: {(a ≡ B) ∧ (b ≡ A)}      ▷ estado final, a e b han intercambiado sus valores iniciales
```

Ejercicio 7 *En ejemplo de la sección 7 referente a la transpuesta de una matriz, hemos empleado dos variables de tipo tabla trans y mat. El programa ha obtenido en trans la matriz*

transpuesta de mat. La matriz mat comieza con unos valores tal y como indica el ejemplo (no nos importa ahora cómo se han dado esos valores, suponemos que ya los tiene), de forma que, para toda posición i y j (en las variables dadas) bien definidas, $mat[i, j] = T_{ij}$ *(cada* T_{ij} *representa el valor inicial). Se trata de hacer un programa en el que la matriz transpuesta se obtenga en la misma variable mat, es decir, al final de la ejecución del programa, para toda posición i y j (bien definidas)* $mat[i, j] = T_{ji}$*. Intenta hacer el ejercicio sin copiar la solución!!.*

Referencias

[1] I. M. Yaglom. Álgebra Extraordinaria. Editorial MIR. Moscú. 1983

[2] Niklaus Wirth. Introducción a la Programación Sistemática. Editorial El Ateneo. Buenos Aires. 1982.

[3] https://www.quantamagazine.org/mathematician-solves-computer-science-conjecture-in-two-pages-20190725/

[4] Hao Huang. Induced subgraphs of hypercubes and a proof of the Sensitivity Conjecture.arXiv:1907.00847; August 2019