Algoritmos y Estructuras de Datos

Apunte **Especificación con TADs**
Segundo Cuatrimestre 2025

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

1. Anatomía de un TAD

Un TAD (Tipo Abstracto de Datos), define un conjunto de valores y las operaciones que se pueden realizar sobre ellos. Es abstracto ya que se enfoca en las operaciones (en cómo interactuar con los datos) y no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones. La especificación de un TAD indica qué hacen las operaciones y no cómo lo hacen.

Empecemos con un ejemplo sencillo:

```
TAD Punto {
    obs x : ℝ
    obs y : ℝ

    proc igualdad(in P_1 : Punto, in P_2 : Punto) : bool {
        asegura { res = true ↔ ··· }
    }

    proc crearPunto(in x : ℝ, in y : ℝ) : Punto {
        asegura { ··· }
    }

    proc mover(inout p : Punto, in dx : ℝ, in dy : ℝ) {
        requiere { ··· }
        asegura { ··· }
    }

    aux theta(p : Punto) : ℝ = ···

    pred estaEnElOrigen(p : Punto)  {
        ···
    }
}
```

Analicemos cada parte:

1.1. Nombre

La primera línea contiene la palabra TAD seguida del nombre del TAD, que debería sea declarativo, es decir, describir el problema a resolver. Luego del nombre tenemos la definición del TAD entre llaves:

```
TAD Punto {
    ···
}
```

1.2. Observadores

```
obs x : ℝ
obs y : ℝ
```

Los observadores permiten describir una instancia del TAD en un determinado momento en términos de tipos más conocidos. La especificación de las operaciones del TAD se escribe en función de los observadores antes y después de su ejecución. Es decir *asegura* deben especificar el valor de todos los observadores. Los tipos de datos que se pueden usar son los de especificación descriptos en el apunte del tema, más dos nuevos tipos complejos $\mathsf{conj}\langle T\rangle$ y $\mathsf{dicc}\langle K, V\rangle$. Todos estos tipos y sus operaciones están descriptos en el Anexo de este apunte.

1.3. Operaciones

Las operaciones de un TAD se especifican mediante nuestro lenguaje de especificación. Se debe indicar el nombre del procedimiento, la lista de parámetros con su nombre, tipo e indicar si son in o inout . Opcionalmente las operaciones pueden devolver un valor. Los parámetros pueden ser de cualquier tipo de nuestro lenguaje de especifiación.

La lista de operaciones que indiquemos en el TAD representan el *contrato* o *interfaz* del TAD, y será lo que luego implementemos y lo que usen los *clientes* del TAD. Por convención, salvo las funciones que crean nuevas instancias del TAD, vamos a usar como primer parámetro una variable de tipo del TAD.

```
proc crearPunto(in x : ℝ, in y : ℝ) : Punto {
   asegura { ··· }
}

proc mover(inout p : Punto, in dx : ℝ, in dy : ℝ) {
   requiere { ··· }
   asegura { ··· }
}
```

Cada función debe indicar la precondición y la postcondición (`requiere` y `asegura`) Como ya indicamos, los requiere y asegura van a hacer referencia a los observadores del TAD. Para referirnos al observador x de la instancia t usamos la notación $t.x$. Para hablar del valor inicial de un parámetro de tipo inout usamos metavariables, refiriendo al tipo que estamos definiendo, no al observador, es decir, $T_0.x$, no $t.X_0$. Las metavariables deben declararse en el `requiere`.

```
proc crearPunto(in x : ℝ, in y : ℝ) : Punto {
   asegura { res.x = x ∧ res.y = y }
}

proc mover(inout p : Punto, in dx : ℝ, in dy : ℝ) {
   requiere { p = P_0 }
   asegura { p.x = P_0.x + dx ∧ p.y = P_0.y + dy }
}
```

Importante: Para que la especificación sea completa, tenemos que describir el valor de todos los observadores al salir de la operación. Recordemos que la implementación final del TAD sólo debe respetar la especificación y puede modificar cualquier cosa que no se defina. Por ejemplo, si tenemos un proc para mover sólo en el eje horizontal x, tenemos que decir que el observador y queda igual:

```
proc moverHoriz(inout p : Punto, in dx : ℝ) {
   requiere { p = P_0 }
   asegura { p.x = P_0.x + dx ∧ p.y = P_0.y }
}
```

1.4. Igualdad

Como parte de la especificación de un TAD, es necesario definir una operación de *igualdad* para definir cuándo dos instancias de un TAD son iguales. Puede alcanzar con que todos los observadores de cada instancia sean iguales a los obs. correspondientes en la otra instancia. Es el caso de nuestro ejemplo: si dos puntos tienen las mismas coordenadas x y y, entonces son el mismo punto. La definición sería

```
proc igualdad(in P1 : Punto, in P2 : Punto) : bool {
    asegura { res = true ↔ P1.x = P2.x ∧ P1.y = P2.y }
}
```

Otras veces la igualdad no es tan directa. Por ejemplo si queremos describir un número racional a partir de dos números enteros (numerador y denominador) tendremos muchas instancias que en realidad representan el mismo número (numerador 1 y denominador 2, num. 4 y den. 8, etc.). En este caso, la *igualdad* debería ser:

```
proc igualdad(in R1 : Racional, in R2 : Racional) : bool {
    asegura { res = true ↔ R1.numerador/R1.denominador = R2.numerador/R2.denominador }
}
```

Para dos instancias t_1 y t_2 de un TAD, si $t_1 = t_2$, entonces toda operación aplicada a t_1 debe dar el mismo resultado que si la aplicamos a t_2, es decir, ambas deben tener el mismo valor de salida y, si aparecen como inout , se debe mantener que $t_1 = t_2$

1.5. Funciones y predicados auxiliares

Para facilitar la especificación, es posible definir predicados y funciones auxiliares, usando nuestro lenguaje de especificación. Estos pueden usarse en los requiere y asegura de las operaciones. Por ejemplo:

```
aux theta(p : Punto) : ℝ = arctan(p.y/p.x)

aux rho(p : Punto) : ℝ = √(p.x² + p.y²)

proc rotar(inout p : Punto, in a : ℝ) {
    requiere { p = P0 }
    asegura {
        p.x = rho(P0) * cos(theta(P0) + a) ∧
        p.y = rho(P0) * sin(theta(P0) + a)
    }
}
```

2. TADs paramétricos

Muchas veces vamos a querer especificar un TAD que tome algún tipo genérico, es decir, indistinto para la especificación, como parámetro. Por ejemplo, podemos definir un conjunto que, como sabemos, va a contener elementos todos de un mismo tipo, y el comportamiento que queremos especificar va a ser el mismo cualquiera sea éste. Definiremos entonces un TAD Conjunto$\langle T \rangle$, donde T representa el tipo de los elementos. Luego, al utilizar el TAD reemplazaremos el tipo T por el tipo concreto que queramos (Conjunto$\langle \mathbb{Z} \rangle$, Conjunto$\langle Punto \rangle$, etc.). A modo de ejemplo, vemos aquí parte de una posible especificación del TAD Cola$\langle T \rangle$.

```
TAD Cola⟨T⟩ {
    obs elems : seq⟨T⟩
        ...
    proc encolar(inout c : Cola⟨T⟩, in e : T) {
        requiere { c = C0 }
        asegura { c.elems = C0.elems ++ ⟨e⟩ }
    }

        ...
}
```

3. Correctitud, modelado y legibilidad

La especificación de problemas es una herramienta para que quede claro qué debe hacer un programa o sistema. Es decir que una especificación debe estar pensada para que quien la lea sepa qué debe implementar a la hora de programar el problema. Con los TADs podemos especificar problemas y sistemas arbitrariamente complejos. Decimos que un TAD es un *modelo* de un problema, es decir, una representación abstracta del comportamiento que debe tener y la información que debe guardar un sistema que lo resuelva.

Un TAD "bien hecho" debería ser

- **Correcto**: Es decir, especificar el problema que queremos resolver, cubriendo todos los casos y las restricciones necesarias.
- **Legible**: Es decir, quien lee el TAD debería poder entender qué problema se está resolviendo sin dificultad.
- Un buen **Modelo** del problema: Es decir, representar la información sobre la que estamos especificando fielmente, sin introducir información extra

3.1. Declaratividad

Para que un TAD sea legible, los nombres que elegimos al definirlo deben ser declarativos, es decir, describir para qué se usa cada observador, procedimiento, predicado, etcétera. Es importante no sólo elegir bien los nombres de los TADs y de los observadores sino también usar *renombres de tipos*.

Por ejemplo, si queremos representar alumnes de una escuela por su número de documento o algún otro identificador numérico podemos escribir.

Alumne `ES` $\mathbb{Z}$

y luego usarlos como cualquier tipo

```
proc calificarAlumne(inout e : Escuela, in a : Alumne, in nota : ℤ) {
   ...
}
```

Cuando necesitamos representar información que tiene varios valores asociados, es más declarativo usar `structs` con nombres en lugar de `tuplas`. Por ejemplo, si queremos representar el alumne con su DNI y nota promedio es más claro

Alumne `ES struct` $\langle DNI : \mathbb{Z}, promedio : \mathbb{R} \rangle$

```
proc promedio(in a : Alumne) : ℝ {
   asegura { res = a.promedio }
}
```

en lugar de

Alumne `ES tupla` $\langle \mathbb{Z}, \mathbb{R} \rangle$

```
proc promedio(in a : Alumne) : ℝ {
   asegura { res = a_1 }
}
```

3.2. Modelado

Los observadores y operaciones de un TAD definen un *modelo* de un problema. Sólo con leerlos deberíamos poder tener una idea de qué es lo importante del problema, qué información está relacionada con cuál y cómo manipularla. Es importante tener esto en cuenta en el momento de la elección de los observadores y de los parámetros de los procedimientos.

Por ejemplo, si queremos representar los alumnes de una escuela, consideremos estos tres observadores alternativos que podríamos elegir:

- `obs` alumnes : conj$\langle Alumne \rangle$
- `obs` alumnes : seq$\langle Alumne \rangle$
- `obs` alumnes : dicc$\langle Alumne, Promedio \rangle$

En el primer caso estamos modelando el alumnado como un grupo de alumnes sin que nos importe más que su DNI (o lo que sea que usamos para identificarlos). En el segundo caso, en cambio, estamos usando una estructura que define un orden[1] entre les alumnes. Finalmente en el tercer caso estamos asociando una nota promedio a cada alumne, con lo que estamos dando a entender que esa nota es importante dentro de nuestro sistema.

En principio ninguno de estos observadores es incorrecto: todos representan a les alumnes de la escuela. Sin embargo, son modelos diferentes, que pueden servir para representar mejor (o peor) diferentes problemas que trabajen con esa información.

Asimismo, consideremos estas opciones de procs:

- `proc abrirEscuela`() : $Escuela$
- `proc abrirEscuela`(in $alumnes$: seq$\langle Alumne \rangle$) : $Escuela$
- `proc abrirEscuela`(in $alumnes$: conj$\langle Alumne \rangle$) : $Escuela$

Las tres son formas válidas de crear el TAD, pero corresponden a modelos distintos. En el primer caso la escuela se "inicializa" sin alumnes (es razonable esperar un `proc inscribirAlumne`(inout e : $Escuela$, in a : $Alumne$)), en el segundo tenemos a todes les alumnes en algún orden, y en el último simplemente a todes les alumnes.

4. Anexo: Tipos de especificación

4.1. Tipos básicos

Las constantes devuelven un valor del tipo. Las operaciones operan con elementos de los tipos y retornan algun elemento de algun tipo. Las comparaciones generan fórmulas a partir de elementos de tipo.

`bool`: valor booleano.

Operación	Sintaxis
constantes	$True$, $False$
operaciones	$\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
comparaciones	$=$, $\neq$

$\mathbb{Z}$: número entero.

Operación	Sintaxis
constantes	1, 2, $\cdots$
operaciones	$+$, $-$, $.$, / (div. entera), mód (módulo), $\cdots$
comparaciones	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

$\mathbb{R}$: número real.

Operación	Sintaxis
constantes	1, 2, $\cdots$
operaciones	$+$, $-$, $.$, /, $\sqrt{x}$, $\sin(x)$, $\cdots$
comparaciones	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

`char`: caracter.

Operación	Sintaxis
constantes	'a', 'b', 'A', 'B'
operaciones	$ord(c)$, $char(c)$
comparaciones (a partir de ord)	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

[1] Además de permitir elementos repetidos, aunque eso se puede resolver en la especificación

4.2. Tipos complejos

4.2.1. tupla $\langle T_1, \ldots, T_n \rangle$

Tupla de tipos T_1, ..., T_n

Operación	Sintaxis
tupla por extensión	$\langle x, y, z \rangle$
obtener un valor	s_i
igualdad	$t_1 = t_2$

- tupla por extensión: permite crear tuplas a partir de sus elementos
  - sintaxis: $\langle x : T_1, y : T_2, \cdots \rangle$: tupla $\langle T_1, T_2, \cdots \rangle$
  - ejemplos: $\langle 1, `hola', 25{,}45 \rangle$ (tupla de tipo $\langle \mathbb{Z}, string, \mathbb{R} \rangle$ con los valores 1, 'hola'y 25.45
- obtener un valor: devuelve el valor en una determinada posición de la tupla. El primer elemento es el 0. Se indefine si el índice es menor a cero o mayor a la cantidad de elementos de la tupla.
  - sintaxis: $\langle T_1, \ldots, T_n \rangle_{i:\mathbb{Z}} : T_i$
  - ejemplos: $\langle 1, `hola', 25{,}45 \rangle_0 = 1$, $\langle 1, `hola', 25{,}45 \rangle_2 = 25{,}45$, $\langle 1, `hola', 25{,}45 \rangle_5$ es indefinido
- igualdad: Resulta *Verdadero* sii t_1 y t_2 tienen
  - misma cantidad de elementos
  - todos del mismo tipo
  - con los mismos valores

4.2.2. struct $\langle campo_1 : T_1, \ldots, campo_n : T_n \rangle$

Tupla con nombres para los campos.

Operación	Sintaxis
struct por extensión	$\langle x : 20, y : 10 \rangle$
obtener el valor de un campo	$s.x, s.y$
igualdad	$s_1 = s_2$

- struct por extensión: permite crear structs a partir de sus campos
  - sintaxis: $\langle campo_1 : T_1, campo_2 : T_2, \cdots \rangle$: struct $\langle campo_1 : T_1, campo_2 : T_2, \cdots \rangle$
  - ejemplos: $\langle x : 20, y : 10 \rangle$ (struct con dos campos de tipo $\mathbb{Z}$ denominados x e y con los valores 20 y 10 respectivamente
- obtener el valor de un campo: devuelve el valor de un campo de la struct. Se indefine si el campo no existe.
  - sintaxis: $\langle campo_1 : T_1, \ldots, campo_n : T_n \rangle.campo_i : T_i$
  - ejemplos: $\langle x : 20, y : 10 \rangle.x = 20$, $\langle x : 20, y : 10 \rangle.z$ es indefinido
- igualdad: Resulta *Verdadero* sii s_1 y s_2 tienen
  - misma cantidad de elementos
  - con los mismos nombres
  - con el mismo tipo para cada nombre
  - con los mismos valores para cada nombre

  Nota: El orden de los campos del struct no es relevante para la igualdad, pero les recomendamos[2] que siempre escriban los campos en el mismo orden.

[2] Les pedimos encarecidamente

4.2.3. $\mathsf{seq}\langle T \rangle$

Secuencia de tipo T

Operación	Sintaxis
secuencia por extensión	$\langle\rangle$, $\langle x, y, z\rangle$
longitud	$\|s\|$, $length(s)$
pertenece	$i \in s$
indexación	$s[i]$
igualdad	$s_1 = s_2$
cabeza	$head(s)$
cola	$tail(s)$
concatenación	$concat(s_1, s_2)$, $s_1 +\!+ s_2$
subsecuencia	$subseq(s, i, j)$
cambiar elemento	$setAt(s, i, val)$

- secuencias por extensión: permite crear secuencias a partir de sus elementos
  - sintaxis: $\langle x : T, y : T, z : T, \cdots\rangle : \mathsf{seq}\langle T\rangle$
  - ejemplos: $\langle\rangle$ (secuencia vacía). $\langle 1, 2, 25\rangle$ (secuencia de enteros con tres elementos: 1, 2 y 25)
- longitud: devuelve la cantidad de elementos de la secuencia
  - sintaxis: $|s : \mathsf{seq}\langle T\rangle| : \mathbb{Z}$. Alternativa: $length(s)$ o $longitud(s)$
  - ejemplos: $|\langle\rangle| = 0$, $|\langle 1, 2, 25\rangle| = 3$
- pertenece: indica si un elemento está en la secuencia
  - sintaxis: $i : T \in s : \mathsf{seq}\langle T\rangle$ (devuelve un valor de verdad)
  - ejemplos: $1 \in \langle\rangle$ es falso, $1 \in \langle 2, 3\rangle$ es falso, $1 \in \langle 1, 2, 3\rangle$ es verdadero
- indexación: devuelve el elemento en una determinada posición de la secuencia. El primer elemento es el 0. Se indefine si el índice es menor a cero o mayor al tamaño de la secuencia.
  - sintaxis: $\mathsf{seq}\langle T\rangle[i : \mathbb{Z}] : T$.
  - ejemplos: $\langle 1, 2, 3\rangle[0] = 1$, $\langle 1, 2, 3\rangle[2] = 3$, $\langle 1, 2, 3\rangle[5]$ es indefinido
- igualdad: Resulta $Verdadero$ sii s_1 y s_2:
  - son del mismo tipo T
  - son del mismo largo
  - para cada posición i, $s_1[i] = s_2[i]$
- cabeza: devuelve el primer elemento de la secuencia. Se indefine si la secuencia es vacía
  - sintaxis: $head(s : \mathsf{seq}\langle T\rangle) : T$
  - ejemplos: $head(\langle 1, 2, 3\rangle) = 1$, $head(\langle\rangle)$ es indefinido
- cola: devuelve la secuencia sin el primer elemento. Se indefine si la secuencia es vacía
  - sintaxis: $tail(s : \mathsf{seq}\langle T\rangle) : \mathsf{seq}\langle T\rangle$
  - ejemplos: $tail(\langle 1, 2, 3\rangle) = \langle 2, 3\rangle$, $tail(\langle 1\rangle) = \langle\rangle$, $tail(\langle\rangle)$ es indefinido
- concatenación: concatena dos secuencias
  - sintaxis: $concat(s_1 : \mathsf{seq}\langle T\rangle, s_2 : \mathsf{seq}\langle T\rangle) : \mathsf{seq}\langle T\rangle$. Alternativa: $s_1 +\!+ s_2$
  - ejemplos: $concat(\langle 1, 2\rangle, \langle 3, 4\rangle) = \langle 1, 2, 3, 4\rangle$, $\langle\rangle +\!+ \langle 1, 2\rangle = \langle 1, 2\rangle$
- subsecuencia: devuelve una subsecuencia de la secuencia original, incluyendo el índice del principio y sin incluir el índice del final. Se indefine si los índices están fuera de rango o si el índice del final es menor al del principio. Devuelve una secuencia vacía si los índices son iguales.
  - sintaxis: $subseq(s : \mathsf{seq}\langle T\rangle, i : \mathbb{Z}, j : \mathbb{Z}) : \mathsf{seq}\langle T\rangle$
  - ejemplos: $subseq(\langle 1, 2, 3, 4, 5\rangle, 1, 3) = \langle 2, 3\rangle$, $subseq(\langle 1, 2, 3, 4, 5\rangle, 1, 1) = \langle\rangle$, $subseq(\langle 1, 2, 3, 4, 5\rangle, 3, 1)$ es indefinido
- cambiar elemento: devuelve una secuencia igual a la original pero con un elemento cambiado. Se indefine si el índice está fuera de rango.
  - sintaxis: $setAt(s : \mathsf{seq}\langle T\rangle, i : \mathbb{Z}, val : T) : \mathsf{seq}\langle T\rangle$
  - ejemplos: $setAt(\langle 1, 2, 3\rangle, 1, 5) = \langle 1, 5, 3\rangle$, $setAt(\langle 1, 2, 3\rangle, 5, 5)$ es indefinido

4.2.4. **string: renombre de** seq⟨char⟩.

4.2.5. conj⟨T⟩

Conjunto de tipo T

Operación	Sintaxis
conjunto por extensión	$\langle\rangle$, $\langle x, y, z\rangle$
tamaño	$\|c\|$
pertenece	$i \in c$
igualdad	$c_1 = c_2$
inclusión	$c_1 \subseteq c_2$
union	$c_1 \cup c_2$
intersección	$c_1 \cap c_2$
diferencia	$c_1 - c_2$

- conjunto por extensión: permite crear conjuntos a partir de sus elementos
  - sintaxis: $\langle x : T, y : T, z : T, \cdots\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\langle\rangle$ (conjunto vacío). $\langle 1, 2, 25\rangle$ (conjunto de enteros con tres elementos: 1, 2 y 25)
- tamaño: devuelve la cantidad de elementos del conjunto
  - sintaxis: $|c : \mathsf{conj}\langle T\rangle| : \mathbb{Z}$. Alternativa: $size(s)$ o $tamaño(s)$
  - ejemplos: $|\langle\rangle| = 0$, $|\langle 1, 2, 25\rangle| = 3$
- pertenece: indica si un elemento está en el conjunto
  - sintaxis: $i : T \in c : \mathsf{conj}\langle T\rangle$ (devuelve un valor de verdad)
  - ejemplos: $1 \in \langle\rangle$ es falso, $1 \in \langle 2, 3\rangle$ es falso, $1 \in \langle 1, 2, 3\rangle$ es verdadero
- igualdad: resulta $Verdadero$ sii c_1 y c_2
  - son del mismo tipo T
  - tienen la misma cantidad de elementos
  - $t \in c_1 \leftrightarrow t \in s_2[i]$
- inclusión: resulta $Verdadero$ sii dados c_1 y c_2
  - son del mismo tipo T
  - todos los elementos de c_1 están en c_2
- unión de conjuntos
  - sintaxis: $c1 : \mathsf{conj}\langle T\rangle \cup c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\langle 1, 2, 3\rangle \cup \langle 1, 4, 5\rangle = \langle 1, 2, 3, 4, 5\rangle$, $\langle 1, 2, 3\rangle \cup \langle\rangle = \langle 1, 2, 3\rangle$
- intersección de conjuntos
  - sintaxis: $c1 : \mathsf{conj}\langle T\rangle \cap c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\langle 1, 2, 3\rangle \cap \langle 1, 4, 5\rangle = \langle 1\rangle$, $\langle 1, 2, 3\rangle \cap \langle\rangle = \langle\rangle$
- diferencia de conjuntos
  - sintaxis: $c1 : \mathsf{conj}\langle T\rangle - c2 : \mathsf{conj}\langle T\rangle : \mathsf{conj}\langle T\rangle$
  - ejemplos: $\langle 1, 2, 3\rangle - \langle 1, 4, 5\rangle = \langle 2, 3\rangle$, $\langle 1, 2, 3\rangle - \langle\rangle = \langle 1, 2, 3\rangle$

4.2.6. $\mathsf{dicc}\langle K, V\rangle$

Diccionario que asocia claves de tipo K con valores de tipo V.

Operación	Sintaxis
diccionario por extensión	$\langle\rangle$, $\langle$"$juan$" : 20, "$diego$" : 10$\rangle$
tamaño	$\|d\|$
pertenece (hay clave)	$k \in d$
igualdad	$d_1 = d_2$
valor	$d[k]$
claves	$claves(d)$
setear valor	$setKey(d, k, v)$
eliminar valor	$delKey(d, k)$

- diccionario por extensión: permite crear diccionaros a partir de sus elementos
  - sintaxis: $\{k_1 : v_1, k_2 : v_2, \cdots\} : \mathsf{dicc}\langle K, V\rangle$
  - ejemplos: {"$juan$" : 20, "$diego$" : 10} (diccionario de tipo K=$string$, V=$\mathbb{Z}$)
- tamaño: devuelve la cantidad de elementos (claves) del diccionario
  - sintaxis: $|c : \mathsf{dicc}\langle K, V\rangle| : \mathbb{Z}$. Alternativa: $size(s)$ o $tamaño(s)$
  - ejemplos: $|\{\}| = 0$, $|\{$"$juan$" : 20, "$diego$" : 10$\}| = 2$
- pertenece: indica si una clave está en el diccionario
  - sintaxis: $k : K \in d : \mathsf{dicc}\langle K, V\rangle$ (devuelve un valor de verdad)
  - ejemplos: "$juan$" $\in \{\}$ es falso, "$juan$" $\in \{$"$juan$" : 20, "$diego$" : 10$\}$ es verdadero
- igualdad: resulta $Verdadero$ sii d_1 y d_2
  - son de los mismos tipos K y V
  - tienen las mismas claves, es decir $\mathsf{claves}(d_1) = \mathsf{claves}(d_2)$
  - todas las claves tienen los mismos valores, es decir para todo k, $d_1[k] = d_2[k]$
- valor: devuelve el valor asociado con una clave. Se indefine si la clave no está en el diccionario
  - sintaxis: $d[k : K] : V$
  - ejemplos: si d = $\langle$"$juan$" : 20, "$diego$" : 10$\rangle$: d["juan"] = 20, d["ana"] está indefinido
- claves: devuelve el conjunto de claves de un diccionario
  - sintaxis: $claves(d : \mathsf{dicc}\langle K, V\rangle) : \mathbb{Z}$
  - ejemplos: si d = $\langle$"$juan$" : 20, "$diego$" : 10$\rangle$: claves(d) = $\langle$"$juan$", "$diego$"$\rangle$
- setKey: Al igual que $setAt$, la función $setKey(d, k, v)$ devuelve un diccionario igual al ingresado pero con el valor de la clave k seteado en v. Es decir, para toda clave k' tal que $k' \in d \vee k' = k$:

$$setKey(d, k, v)[k'] = \begin{cases} v & \texttt{si } k' = k \\ d[k'] & \texttt{si } k' \neq k \end{cases}$$

  - sintaxis: $setKey(d : \mathsf{dicc}\langle K, V\rangle, k : K, v : V) : \mathsf{dicc}\langle K, V\rangle$
  - ejemplos: si d = $\langle$"$juan$" : 20, "$diego$" : 10$\rangle$: $setKey(d,$ "$juan$", $5) = \langle$"$juan$" : 5, "$diego$" : 10$\rangle$
- delKey: La función $delKey(d, k)$ elimina una clave del diccionario y deja igual todo el resto de los valores.
  - sintaxis: $delKey[d : \mathsf{dicc}\langle K, V\rangle, k : K) : \mathsf{dicc}\langle K, V\rangle$
  - ejemplos: si d = $\langle$"$juan$" : 20, "$diego$" : 10$\rangle$: $delKey(d,$ "$juan$"$) = \langle$"$diego$" : 10$\rangle$

4.3. Sumatorias y productorias sobre conjuntos y diccionarios

Se puede escribir sumatorias y productorias sobre conjuntos usando la siguiente sintáxis

$$elementosPositivos = \sum_{e \in c} \big(\mathsf{IfThenElse}(e \geq 0, 1, 0)\big)$$

Para hacer algo similar sobre un diccionario hay que usar el diccionario de claves

$$significadosPositivos = \sum_{c \in claves(d)} \big(\mathsf{IfThenElse}(d[c] \geq 0, 1, 0)\big)$$