Lenguaje de Especificación

Contenido

Tipos de datos

Recordemos que un tipo de datos es un conjunto de valores con ciertas operaciones en común. Vamos a empezar viendo unos tipos de datos básicos para después introducir otros más complejos.

En nuestro lenguaje de especificación, para hablar de un elemento de un tipo, escribimos un **término o expresión**. Un término puede ser

- una variable,
- una constante del tipo
- o una función (operación) aplicada a otros términos.

Todos los tipos tienen un elemento distinguido (un valor especial, distinto de los demás) llamado **indefinido**. En el lenguaje de especificación, podemos representarlo por la constante `Indef` o por el símbolo $\perp$.

Tipos de Datos Básicos

Valores de verdad (`Bool`)

Es un tipo muy importante, porque sus términos son los **predicados** (afirmaciones, condiciones) de nuestro lenguaje de especificación. Por ejemplo, la precondición y la poscondición de un problema son términos de tipo `Bool`.

Este tipo tiene dos constantes: `True` y `False`.

Todos los tipos del lenguaje tienen una operación *a* `==` *b*, que es de tipo `Bool` e indica si los términos *a* y *b* representan el mismo valor. La operación contraria se escribe *a* `!=` *b* o *a* $\neq$ *b*.

Las operaciones más frecuentes de tipo `Bool` son los conectivos lógicos:

- $a_1 \wedge a_2 \wedge \ldots \wedge a_n$: conjunción (se puede escribir && en lugar de $\wedge$),
- $a_1 \vee a_2 \vee \ldots \vee a_n$: disyunción (se puede escribir || en lugar de $\vee$),
- $\neg a$: negación (también se puede escribir `no(`*a*`)`),
- $a \rightarrow b$: implicación (también se puede escribir *a* `-->` *b*).

La igualdad del tipo `Bool` (*a* `==` *b*) es la doble implicación (si y solo si), que se puede escribir también `<-->` o $\leftrightarrow$.

Los conectivos tienen **semántica secuencial** o sea que los términos se evalúan de izquierda a derecha y la evaluación termina cuando se puede deducir el valor de verdad de la expresión completa, aunque el resto esté indefinido. Por lo tanto,

- `(False &&` *a*`) == False`
- `(True ||` *a*`) == True`
- `(False ->` *a*`) == True`

En estos ejemplos, *a* puede ser cualquier término de tipo `Bool`, con valor `True`, `False` o `Indef`.

Números enteros (`Int`)

El tipo `Int` corresponde al conjunto **Z** (el lenguaje también permite escribirlo con este símbolo), de los números enteros.

Las constantes del tipo son los literales que representan números enteros en notación decimal: `0`; `1`; `-1`; `2`; `-2`; `3`; `-3`; ...

Sus funciones son las operaciones más comunes en **Z**:

- *a* + *b*,
- *a* – *b*,
- *a* * *b* (multiplicación, también se puede escribir $a \cdot b$ o $a \times b$),
- –*a*,
- `abs(`*a*`)` (valor absoluto),
- *a* `div` *b* (división entera, $b \neq 0$),
- *a* `mod` *b* (resto de la división entera, $b \neq 0$),
- `pot(`*a*`,`*b*`)` (potenciación, también se puede escribir a^b).

Las comparaciones entre enteros son términos de tipo `Bool`:

- *a* < *b*,
- *a* > *b*,
- *a* <= *b* (o $a \leq b$),
- *a* >= *b* (o $a \geq b$).

Las comparaciones se pueden encadenar: si escribimos $a_1\ \zeta_1\ a_2\ \zeta_2\ a_3$, donde los ζ_i son comparaciones (<, >, <=, >=, ==), es como si hubiéramos escrito $(a_1\ \zeta_1\ a_2) \wedge (a_2\ \zeta_2\ a_3)$, y así sucesivamente para cadenas más largas.

La operación `beta(`*a*`)` se aplica a un término de tipo `Bool` y devuelve su valor numérico de verdad (`0` o `1`). Por ejemplo `beta(1 < 2) == 1`, y `beta(2 < 1) == 0`. Otra forma de escribirla es $\beta(a)$.

Números reales (`Float`)

Sus valores son los del conjunto **R** (que también es una notación válida para el tipo).

Sus constantes son literales que representan números racionales escritos en base diez, usando el punto como separador de la parte decimal: `0`; `1`; `5`; `2.7`; `-92.896`; ... También vamos a usar la constante `Pi` (o π). Como pueden ver, algunas de estas constantes son compartidas con el tipo `Int`, pero esto no ocasiona problemas, porque el tipo al que pertenecen se puede deducir del contexto.

El tipo `Float` soporta las mismas operaciones que `Int`, excepto `div` y `mod`. Se agregan además

- la división (*a* `/` *b*),
- el logaritmo de b en base a (`log(`*a*`,`*b*`)` o $\log_a b$),
- las funciones trigonométricas (`sin`, `cos`, `tan`, `atan`, etc.).

La operación `int(`*a*`)` devuelve la parte entera de un real (el resultado es de tipo `Int`), también se escribe $\lfloor a \rfloor$. Los términos de tipo `Int` pueden usarse como si fueran de tipo `Float`, prestando atención al tipo del resultado de las operaciones (por ejemplo, la división entre dos enteros es siempre un término de tipo `Float`).

Caracteres (`Char`)

Cada elemento de este tipo es una letra o símbolo. Sus constantes se escriben entre comillas simples: `'A'`, `'B'`, ..., `'Z'`, ..., `'a'`, `'b'`, ..., `'z'`, ..., `'0'`, `'1'`, ..., `'9'`, ...

La función `ord(a)` numera todos los caracteres (devuelve un número entero distinto para cada carácter). No es importante qué número corresponde a cada carácter, pero sí se sabe que la numeración es coherente con el orden alfabético y numérico:

- `ord('A') + 1 == ord ('B');`
- `ord('a') + 1 == ord ('b');`
- `ord('1') + 1 == ord ('2').`

La función inversa es `char(`*a*`)`: dado un entero, devuelve el carácter correspondiente, y también se puede escribir $\texttt{ord}^{-1}(a)$.

Las comparaciones entre caracteres se interpretan como comparaciones entre sus órdenes: $(a < b) == (\texttt{ord}(a) < \texttt{ord}(b))$.

Términos

Los términos de un tipo pueden ser **términos simples** (variables o constantes del tipo) o **términos compuestos** (combinaciones de funciones aplicadas a términos de los tipos correspondientes).

Por ejemplo, los siguientes son términos de tipo `Int`:

- `0 + 1`
- `pot(((3 + 4) * 7),2)`
- `– 1`
- `2 * β(1 + 1 == 2)`
- `1 + ord('A')`

Y estos otros también lo son, siempre y cuando `x` sea una variable de tipo `Int`; `y`, de tipo `Float`; `z` de tipo `Bool`:

- `2 * x + 1`
- `β(pot(y,2) > π) + x`
- `(x mod 3) * β(z)`

En cuanto a su semántica, los términos representan elementos del tipo correspondiente. Su valor es `Indef` cuando no se puede realizar alguna de las operaciones que contiene. Por ejemplo,

- `1 div 0`
- `pot(-1, 0.5)`

Todas las operaciones se consideran **estrictas** (si uno de sus argumentos es indefinido, el resultado también lo será); excepto los conectivos lógicos, como ya hemos visto. Por ejemplo, el término `0 * (1/0)` es indefinido. En particular, intentar comparar la igualdad de un término con otro indefinido es una expresión indefinida. Esto vale también para los casos en que ambos términos son indefinidos; por ejemplo, `Indef == Indef` es `Indef` y no `True`.

Funciones Auxiliares

Para simplificar la escritura y la lectura de especificaciones, muchas veces vamos a definir funciones auxiliares. Se trata de funciones que se pueden usar (únicamente) en la especificación, a diferencia de las que especificamos como solución a un problema. El mecanismo para definirlas simplemente le asigna un nombre a una expresión.

La sintaxis es la siguiente:
`aux` *f*`(`*parámetros*`):` *tipo* `=` *e*`;`

Donde *f* es el nombre que se le quiere dar a la función auxiliar, que podrá usarse en el resto de la especificación en lugar de la expresión *e*. Los *parámetros* (opcionales) pueden aparecer en *e* y se reemplazan por las expresiones correspondientes cada vez que se utiliza *f*.

Por ejemplo, si definimos `aux suc(i: Int): Int = i + 1;` podemos después usar la función `suc` en la poscondición de un problema.

tipo es el tipo resultante de la función definida y debe coincidir con el tipo de la expresión que la define (*e*).

Es importante tener clara la diferencia entre **definir** una función auxiliar y **especificar** un problema. Cuando definimos una función auxiliar, damos una expresión del lenguaje a la que la función es equivalente. Y esto nos permite usar la función dentro de las especificaciones. Cuando especificamos un problema, en cambio, estamos dando las condiciones (el contrato) que debería cumplir alguna función para ser solución del problema. Eso no quiere decir que exista esa función (podría no haber ningún algoritmo que sirviera como solución) o que sepamos cómo escribirla. Si alguna vez damos una función que solucione el problema planteado, vamos a hacerlo en otro lenguaje (probablemente, en un lenguaje de programación), no en el lenguaje de especificación. Por todo esto no podemos usar, en la especificación de un problema o de un tipo, otra función que hayamos especificado.

Tipos Enumerados y Sinónimos de Tipo

Los tipos enumerados son el primer mecanismo que van a aprender para definir sus propios tipos de datos. Son tipos que tienen una cantidad finita de elementos, representados cada uno por una constante distinta. Para declarar un tipo enumerado en una especificación, escribimos
`tipo` *Nombre* `=` *constantes*`;`

El *Nombre* que le damos al tipo tiene que ser distinto de todos los existentes. Las constantes son también nombres nuevos, separados por comas. Por convención, todos estos nombres empiezan con mayúscula.

Las constantes son términos del tipo. Se cuenta con una operación `ord(`*a*`)` que devuelve la posición (empezando de `0`) del elemento en la lista de definición. Su opuesta es *nombre*`(`*a*`)`, que también se puede escribir $\texttt{ord}^{-1}$ (hay que tener cuidado al usarla, porque esta función no está definida para valores mayores o iguales a la cantidad de elementos del tipo). Las comparaciones se definen de acuerdo al orden, como hemos visto para el tipo `Char`.

Por ejemplo, si definimos
`tipo Día = Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo;`
valen `ord(Lunes) == 0`; `Día(2) == Miércoles`; `Jueves < Viernes`.

Y podemos definir, por ejemplo, una función auxiliar que diga si un día pertenece al fin de semana:
```
aux esFinde(d: Día): Bool = d == Sábado || d == Domingo;
```

O se puede hacer así:
```
aux esFinde2(d: Día): Bool = d > Viernes;
```

Los sinónimos de tipo sirven simplemente para asignar otro nombre a un tipo. En el resto de la especificación, pueden usarse ambos nombres indistintamente:
`tipo Edad = Int;`

Secuencias

Las secuencias (también se las llama **listas**) son una **familia de tipos** que tiene un lugar muy importante en el lenguaje de especificación. Son una familia porque para cada tipo de datos existe un tipo secuencia correspondiente, que tiene como elementos las secuencias de elementos de ese tipo.

Las secuencias están formadas por varios elementos del mismo tipo, posiblemente repetidos, ubicados en un cierto orden.

El tipo de las secuencias con elementos de tipo T se llama `[`T`]` (se lee "secuencia de T"). Y una forma de escribir un elemento de este tipo es escribiendo entre corchetes varios términos de tipo T separados por comas. Por ejemplo, el siguiente es un término de tipo `[Int]` (secuencia de `Int`s):
`[1,1+1,3,2*2,3,5]`

La secuencia vacía (de elementos de cualquier tipo) se representa con dos corchetes enfrentados: `[]`.

Se puede formar secuencias de elementos de cualquier tipo. Como las secuencias son tipos, en particular se puede usar secuencias de secuencias. Por ejemplo, el siguiente es un término del tipo `[[Float]]` (secuencia de secuencias de `Float`s):
`[[2.3, 7.1], [23.6, 9, 0], [5], [], [], [4.3]]`

Secuencias por comprensión

La notación de secuencias por comprensión es una manera de construir secuencias de elementos que cumplan ciertas condiciones, a partir de otras secuencias existentes.

[*expresión* | *selectores*, *condiciones*]

expresión: cualquier expresión válida del lenguaje, suele contener las variables de los selectores.

selectores: son de la forma *variable* ∈ *secuencia*, y puede haber varios separados por coma (también se puede usar <- en lugar de ∈). La *variable* irá tomando, por turno, el valor de cada uno de los elementos de la *secuencia*. Las variables que aparecen en selectores se llaman **variables ligadas**. El resto de las variables de una expresión se llaman **variables libres**.

condiciones: expresiones de tipo `Bool`; suelen contener las variables de los selectores. Se separan por comas y se consideran relacionadas por el operador &&.

El resultado es una secuencia con el valor de la *expresión* calculado para todos los elementos seleccionados por los *selectores* que cumplen las *condiciones*.

Ejemplo:

```
[(x,y) | x ∈ [1,2], y ∈ [1,2,3], x < y] == [(1,2),(1,3),(2,3)]
```

[$expresión_1$..$expresión_2$]

Define una secuencia mediante un intervalo. Las expresiones son del mismo tipo, discreto y totalmente ordenado (por ejemplo, `Int`, `Char`, enumerados). El resultado son todos los valores del tipo entre el de la $expresión_1$ y el de la $expresión_2$ (ambas inclusive).

Si no vale $expresión_1$ `<=` $expresión_2$, la secuencia es vacía.

Usando paréntesis en lugar de corchetes, se puede excluir uno o los dos extremos.

Ejemplos:

```
[5..9] == [5,6,7,8,9]
[5..9) == [5,6,7,8]
(5..9] == [6,7,8,9]
(5..9) == [6,7,8]
```

Operaciones frecuentes

En esta sección presentamos algunas operaciones con secuencias que usaremos frecuentemente en las especificaciones. Más adelante examinaremos las secuencias con más detalle. Muchas de estas operaciones tienen una precondición, porque no pueden aplicarse a cualquier combinación de argumentos, si la precondición no se cumple el resultado es indefinido. Esto no significa que estén especificadas mediante contratos: luego veremos que se trata de observadores y funciones auxiliares.

Longitud: `long(a: [T]): Int`

Es la longitud de la secuencia `a`.

Notación: `long(a)` se puede escribir `|a|`.

Indexación: `índice(a: [T], i: Int): T`

`requiere 0 <= i < |a|;`

Es el elemento en la `i`-ésima posición de `a`. La primera posición es la `0`.

Notación: `índice(a,i)` se puede escribir `a[i]` y también a_i.

Cabeza: `cab(a: [T]): T`

`requiere |a| > 0;`

Es el primer elemento de la secuencia.

Cola: `cola(a: [T]): [T]`

`requiere |a| > 0;`

Es la secuencia sin su primer elemento.

Pertenencia: `en(t: T, a: [T]): Bool`

Indica si el elemento aparece (al menos una vez) en la secuencia.

Notación: `en(t,a)` se puede escribir `t en a` y también `t` $\in$ `a`. Otra forma de escribir `no(t en a)` es `t` $\notin$ `a`.

Agregar cabeza: `cons(t: T, a: [T]): [T]`

Construye una secuencia como la recibida, a la que se le ha agregado `t` como primer elemento.

Notación: `cons(t,a)` se puede escribir `t:a`.

Concatenación: `conc(a, b: [T]): [T]`

Construye una secuencia con los elementos de `a`, seguidos de los de `b`.

Notación: `conc(a,b)` se puede escribir `a ++ b`.

Subsecuencia: `sub(a: [T], d, h: Int): [T]`

La subsecuencia de `a` formada por los elementos ubicados en las posiciones entre `d` y `h` (ambos inclusive).

Todos los casos en que no se cumple `0 <= d <= h < |a|` equivalen a la secuencia vacía.

Notación para obtener subsecuencias mediante intervalos.

```
a[d..h] == sub(a,d,h)
a[d..h) == sub(a,d,h-1)
a(d..h] == sub(a,d+1,h)
a(d..h) == sub(a,d+1,h-1)
a[d..] == sub(a,d,|a|-1)
a(d..] == sub(a,d+1,|a|-1)
a[..h] == sub(a,0,h)
a[..h) == sub(a,0,h-1)
```

En las especificaciones resulta muy frecuente usar intervalos abiertos a derecha, ya que `|a[d..h)| == h - d`, y `|a[..h)| == h`.

Asignación a una posición: `cambiar(a: [T], i: Int, val: T): [T]`

`requiere 0 <= i < |a|;`

Es una secuencia igual a la secuencia `a`, excepto porque el valor en la posición `i` es `val`.

Operaciones de combinación

Estas operaciones combinan los valores de todos los elementos de una secuencia de valores de verdad o de números.

Todos verdaderos: `todos(sec: [Bool]): Bool`

Es verdadero solamente si todos los elementos de la secuencia son `True` (o la secuencia es vacía).

Alguno verdadero: `alguno(sec: [Bool]): Bool`

Es verdadero solamente si algún elemento de la secuencia es `True` (y ninguno es `Indef`).

Sumatoria: `sum(sec: [T]): T`

T debe ser un tipo numérico (`Float`, `Int`).

Calcula la suma de todos los elementos de la secuencia. Si la secuencia es vacía, el resultado es `0`.

Notación: `sum(sec)` se puede escribir `Σ sec`.

Ejemplo:

```
aux(n: Int) PotenciasNegativasDeDosHastan: Float = Σ [2^-m | m ∈ [1..n]];
```

Productoria: `prod(sec: [T]): T`

T debe ser un tipo numérico (`Float`, `Int`).

Calcula el producto de todos los elementos de la secuencia. Si la secuencia es vacía, el resultado es `1`.

Notación: `prod(sec)` se puede escribir `Π sec`.

Para todo: `(∀` *selectores*, *condiciones*`)` *expresión*

Es un término de tipo `Bool` que permite afirmar que todos los elementos de una lista definida por comprensión cumplen una propiedad (representada por la *expresión*, que también debe ser `Bool`). Es simplemente una forma equivalente de escribir lo siguiente:

`todos([`*expresión* `|` *selectores*, *condiciones*`]).`

Ejemplo (los elementos en posiciones pares son mayores que `5`):

```
aux par(n: Int): Bool = n mod 2 == 0;
```

```
aux posParM5(a: [Int]): Bool = (∀ i ∈ [0..|a|), par(i)) a[i]>5;
```

La expresión que define esta función es equivalente a esta otra:

```
todos([a[i]>5 | i ∈ [0..|a|), par(i)]);
```

Notación: en lugar de ∀ se puede escribir `paratodo`.

Existe: (∃ *selectores*, *condiciones*) *expresión*

La diferencia con la notación anterior es que, en lugar de afirmar que *todos* los elementos de la lista determinada por los *selectores* y *condiciones* cumplen la *expresión*, aquí se dice que hay *alguno* que la cumple. Es, por lo tanto, equivalente a escribir lo siguiente:

`alguno([`*expresión* `|` *selectores*, *condiciones*`]).`

Ejemplo (hay algún elemento de la lista que es par y mayor que 5):

```
aux hayParM5(a: [Int]): Bool = (∃ x ∈ a, par(x)) x>5;
```

Y es equivalente a definir

```
aux hayParM5'(a: [Int]): Bool = alguno([x>5 | x ∈ a, par(x)]);
```

Notación: en lugar de ∃ se puede escribir `existe` o `existen`.

Cantidades

A veces queremos saber cuántos elementos de una secuencia cumplen una condición. Una forma de hacerlo es contando la cantidad de elementos de una secuencia construida por comprensión. Por ejemplo, para ver cuántas veces aparece el elemento `x` en la secuencia `a`, podemos definir esta función:

```
aux cuenta(x: T, a: [T]): Int = long([y | y <- a, y == x]);
```

Y podemos usarla para ver si dos secuencias tienen los mismos elementos, aunque en distinto orden:

```
aux mismos(a, b: [T]): Bool = |a| == |b| ∧ (∀ c ∈ a) cuenta(c,a) == cuenta(c,b);
```

Otro ejemplo:

```
aux cantPrimosMenores(n: Int): Int = long([y | y <- [0..n), primo(y)]);
aux primo(n: Int): Bool = n >= 2 && ¬ (∃ m ∈ [2..n)) n mod m == 0;
```

Acumulación

La notación de acumulación provee un mecanismo similar al de construcción de secuencias por comprensión, aumentando su poder expresivo. Construye un valor a partir de una o más secuencias. La forma general de una expresión de acumulación es la siguiente:

`acum(`*expresión* `|` *inicialización*, *selectores*, *condición*`)`

La diferencia entre esta sintaxis y la de secuencias por comprensión está en la inicialización, que es de la forma *acumulador*: *tipoAcum* = *init*. Donde *acumulador* es una variable e *init* es una expresión de tipo *tipoAcum*.

El resto de los componentes coinciden con los de las secuencias por comprensión, excepto porque en *expresión*, además de las variables de los selectores, puede aparecer el *acumulador*. El *acumulador* solo puede aparecer en la *expresión* pero no en la *condición*.

El resultado de la acumulación es de tipo *tipoAcum*.

El *acumulador* tiene como valor inicial el de *init*. A medida que las variables de los *selectores* toman cada uno de los valores de la secuencia correspondiente, se

vuelve a calcular el valor de la *expresión* y ese es el valor que adquiere el *acumulador*. El resultado de la acumulación es el valor final del *acumulador*.

Veamos algunos ejemplos:

```
aux sum(l: [Float]): Float = acum(s + i | s: Float = 0, i <- l);
aux prod(l: [Float]): Float = acum(p * i | p: Float = 1, i <- l);
```

La siguiente función auxiliar construye, dado `n` $\geq$ 1, la sucesión de los `n+1` primeros números de la de Fibonacci:

```
aux fiboSuc(n: Int): [Int] =
                    acum(f ++ [f[i-1]+f[i-2]] | f: [Int] = [1,1], i <- [2..n]);
```

Observar que para `n` = 0, el aux anterior es la lista `[1,1]`.

Y esta otra calcula el `n`-ésimo número de Fibonacci, para `n` $\geq$ 1:

```
aux fibo(n: Int): Int = (fiboSuc(n-1))[n-1];
```

Esta otra función concatena todos los componentes de una secuencia de secuencias. Tiene el efecto de construir una secuencia "aplanada":

```
aux concat(a: [[T]]): [T] = acum(l ++ c | l: [T] = [], c <- a);
```

La construcción de secuencias por comprensión puede definirse en términos de la acumulación. El término `[`*expresión* `|` *selectores*, *condición*`]`, donde *expresión* es de tipo *T* es equivalente a

`acum(res ++ [`*expresión*`] | res: [`*T*`] = [], `*selectores*, *condición*`)`.

Especificación de Problemas

Recordemos que vamos a especificar problemas que se resuelvan mediante funciones. La función que sirva como solución va a tener por conjunto de partida los valores posibles de sus parámetros. Siempre vamos a considerar que los parámetros tienen valores definidos y tamaño finito. La especificación determina el contrato que debería cumplir la función para ser considerada solución del problema.

Como también mencionamos, es importante distinguir el *qué* (la especificación, el contrato a cumplir) del *cómo* (la implementación de la función, que se escribirá luego en un lenguaje de programación). Es decir que en la especificación solamente vamos a decir qué condiciones tiene que cumplir la solución, pero vamos a evitar referencias a posibles métodos para resolver el problema.

Esta técnica de especificación deja abierta la posibilidad a dar distintas soluciones a un mismo problema. Una diferencia obvia entre una solución y otra puede ser el lenguaje de programación elegido para escribir la función. En la materia vamos a trabajar con dos; y se van a dar cuenta de que no solamente difieren en su sintaxis, sino también en la manera en que se piensan las soluciones para implementarlas en cada uno de ellos.

Incluso si se elige un único lenguaje de programación, generalmente va a haber distintas formas de programar una función que cumpla con la especificación (si es que hay alguna, como veremos más adelante). Cada algoritmo posible constituye una solución distinta para el problema. En consecuencia, para cada especificación existe un conjunto (tal vez vacío) de algoritmos que la cumplen.

Ejemplos

Veamos algunos ejemplos de especificación de problemas.

- Si el problema que tenemos es calcular el cociente de dos enteros dados, podríamos escribir la siguiente especificación:

```
problema división(a, b: Int) = result: Int {
   requiere b != 0;
   asegura result == a div b;
}
```

- Extendamos ahora el problema a calcular el cociente y el resto de la división, siempre y cuando el divisor sea positivo. Se nos presenta entonces la necesidad de que la función devuelva dos valores. Una forma de lograrlo es usando un par ordenado. Las **tuplas** son tipos de datos que veremos en detalle más adelante. Por ahora, alcanza con saber que el tipo `(Int, Int)` representa los pares ordenados de enteros y que las funciones `prm` y `sgd` devuelven la primera y la segunda componente, respectivamente.

```
problema cocienteResto(a, b: Int) = result: (Int,Int) {
   requiere b > 0;
   asegura a == q * b + r && 0 <= r < b;
   aux q: Int = prm(result), r: Int = sgd(result);
}
```

  Como la poscondición se hacía un poco larga, les pusimos nombre a las dos componentes del resultado. También noten que definimos dos nombres en una única sentencia `aux`.

  Una alternativa hubiera sido dejar el cociente como único resultado del problema, y agregarle un parámetro modificable para representar el resto. En matemática, la aplicación de funciones da un resultado, sin modificar el valor de los parámetros; sin embargo nuestro lenguaje permite especificar que la solución a un problema sí lo hace.

```
problema cocienteResto2(a, b, r: Int) = q: Int {
   requiere b > 0;
   modifica r;
   asegura a == q * b + r && 0 <= r < b;
   aux q: Int = prm(result), r: Int = sgd(result);
}
```

  Podríamos haber usado también un parámetro para devolver el cociente; en cuyo caso el problema no tiene ningún resultado:

```
problema cocienteResto3(a, b, q, r: Int) {
   requiere b > 0;
   modifica q, r;
   asegura a == q * b + r && 0 <= r < b;
   aux q: Int = prm(result), r: Int = sgd(result);
}
```

- Planteémonos ahora, el problema de calcular la suma de los inversos multiplicativos de varios números reales. Como no sabemos cuántos números van a ser (la cantidad va a variar en cada instancia del problema), no podemos usar tuplas, que tienen una cantidad fija de componentes. Vamos a trabajar entonces con secuencias:

```
problema sumarInvertidos(a: [Float]) = result: Float {
   requiere 0 ∉ a;
   asegura result == Σ [1/x | x <- a];
}
```

En la precondición estamos pidiendo que el argumento no contenga ningún `0` (porque ese número no puede invertirse). Si no lo pidiéramos, la poscondición podría indefinirse, y esta es una situación que tenemos que evitar a toda costa: **las precondiciones y las poscondiciones deben estar definidas para cualquier valor de los parámetros**. Podríamos haber escrito esta misma precondición de distintas formas, las siguientes son todas equivalentes:

```
requiere (∀ x ∈ a ) x ≠ 0;
requiere ¬(∃ x ∈ a) x == 0;
requiere ¬(∃ i ∈ [0..|a|)) a[i] == 0;
requiere [x | x <- a, x == 0] == [];
```

- El siguiente problema a especificar es encontrar una raíz de un polinomio de grado 2 a coeficientes reales. Digamos que el polinomio va a venir descrito por sus coeficientes `a`, `b` y `c`, en grado decreciente:

```
problema unaRaízPoli2(a, b, c: Float) = r: Float {
   asegura a * r * r + b * r + c == 0;
}
```

Esta especificación no tiene precondición, lo cual es equivalente a decir que la precondición es `True`. Si construyéramos una función que resolviera este problema y la llamáramos con argumentos `1`, `0`, `1`; obtendríamos como resultado `r == unaRaízPoli2(1,0,1)` tal que `r*r + 1 == 0`. Pero `r` sería un real cuyo cuadrado es `-1`, lo cual no existe. Este es un ejemplo de especificación que no puede ser cumplida por ninguna función. Es un problema que no tiene solución.

El error estuvo en la escritura de la especificación: **no es correcto escribir especificaciones que no puedan cumplirse**. La precondición es demasiado débil. Deberíamos garantizar que el polinomio tuviera raíces reales. La nueva precondición podría ser (entre otras) alguna de las que siguen:

```
requiere b*b >= 4*a*c;
```

En cuanto a la poscondición, sirve de ejemplo para ver cómo la especificación debe describir qué hacer y no cómo hacerlo. No dice cómo calcular la raíz, ni qué raíz devolver. El siguiente es un ejemplo de **sobrespecificación**: una poscondición que pone más restricciones de las necesarias, al punto que fija la forma de calcular la solución:

```
asegura result == (-b+(b²-4*a*c)^1/2)/(2*a)
```

- El próximo problema es encontrar el índice (la posición) del elemento de menor valor en una secuencia de números reales distintos no negativos:

```
problema índiceMenorDistintos(a: [Float]) = res: Int {
   requiere NoNegativos: todos([x >= 0 | x <- a]);
   requiere Distintos: (∀ i∈[0..|a|), j∈[0..|a|), i≠j) a_i ≠ a_j;
   asegura 0 <= res < |a|;
   asegura (∀ x <- a) a_res <= x;
}
```

En este ejemplo elegimos ponerles nombres a las precondiciones para aclarar su significado. Estos nombres también pueden usarse como predicados en cualquier lugar de la especificación. Atención: el nombre de los predicados es solamente eso: una forma de hacer referencia a ellos. Por más que escribamos `requiere NoNegativos`, si no ponemos luego una condición que efectivamente ponga esa restricción explicando en el lenguaje de especificación qué significa, el problema

podrá recibir valores negativos. También aparecen en el ejemplo varias notaciones alternativas combinadas, para que vayan familiarizándose con ellas.

Otra forma de escribir la segunda precondición:

```
requiere Distintos2: (∀ i∈[0..|a|)) a[i] ∉ a[..i);
```

Sintaxis

La especificación de un problema tiene esta forma:

problema *nombre* (*parámetros*) = *nombreRes*: *tipoRes contrato*

parámetros es una lista opcional, separada por comas, donde cada elemento es de la forma *variable*: *tipo*, o simplemente una variable. En este último caso, el tipo será el del parámetro siguiente (el último elemento debe incluir su tipo).

tipoRes es el tipo del resultado y *nombreRes* es el nombre con el que vamos a referirnos a ese resultado en el contrato. Cuando el problema modifica sus parámetros en lugar de dar un resultado, se omiten ambos (y el `=`).

El *contrato* está formado por varias **sentencias** encerradas entre llaves (`{` y `}`). Las sentencias está encabezada por una palabra clave (`requiere`, `asegura`, `modifica` o `aux`), como se explica a continuación.

Todas las sentencias terminan con un punto y coma que es opcional.

requiere *nombre*: *P*;

Introduce una precondición en la especificación de una función (problema). El predicado *P* debe cumplirse antes de la invocación de la función. Equivale a la obligación del usuario en el contrato especificado.

Si hay más de una precondición, se las supone unidas por el operador `&&`, con su evaluación de cortocircuito (es decir que si una es falsa, no importa que las siguientes se indefinan).

Cuando se resuelva el problema se hará mediante una función, que es lo que se llama una **implementación** de la especificación. Todas las precondiciones pueden suponerse ciertas en la implementación, porque si no valen cuando se la invoca, la función no está obligada a cumplir el contrato. Esta semántica es de *eximición de responsabilidad*; difiere de una semántica de *condición de activación*, que significaría que la función no se va a ejecutar si se la llama en un estado que no satisfaga la precondición.

Por ejemplo, considérese la especificación de la raíz cuadrada que pide la siguiente precondición: `requiere x >= 0;`

Si el valor del argumento es negativo, la especificación no indica qué debería hacer la implementación, por lo que esa posibilidad no necesita considerarse al escribir esta última.

Si una precondición es el predicado constante `True`, se puede omitir la cláusula `requiere` completa. Estas funciones pueden ser llamadas siempre. Su implementación no puede hacer suposiciones sobre la llamada, excepto la que consiste en que los parámetros van a tener los tipos declarados (y que tienen valores definidos, de tamaño finito).

El *nombre* es optativo y puede servir tanto para aclarar el objetivo de la precondición, como para hacer referencia a ella en otros predicados.

asegura *nombre*: *P*;

Introduce una poscondición en la especificación de un problema. *P* es generalmente un predicado en el que se mencionan los parámetros y el resultado del problema. Representa las obligaciones, en el contrato especificado, de una función que resuelva el problema. Cuando se razona sobre un llamado a esa función, puede suponerse cierta luego de la invocación.

Si hay más de una poscondición, se las evalúa en orden como si estuvieran unidas con el operador &&.

La interpretación de una especificación es que si las precondiciones se cumplen; entonces la función que resuelva el problema debe terminar normalmente, en un estado que cumpla las poscondiciones.

En *P* puede aparecer nombre que se le dio al resultado en el encabezado, representando al valor calculado por la función.

modifica *variables*;

Introduce la lista de *variables* que pueden ser modificadas por la ejecución de la función. Una variable se modifica si su valor cambia como efecto de la ejecución de la función especificada (tiene un valor distinto en el pre-estado y en el post-estado). Nótese que la cláusula `modifica` no significa que la ejecución *deba* cambiar los valores de las variables; simplemente le otorga a la función permiso para cambiarlas. Por ejemplo, si una función que intercambia los valores de dos de sus parámetros, recibe dos variables con el mismo valor, no debería haber ningún cambio.

La cláusula `modifica` es un atajo sintáctico para evitar escribir un predicado más largo en la poscondición. Como solamente los objetos listados en la cláusula `modifica` pueden cambiar de valor en la ejecución, los demás deben mantenerlo.

Cuando la especificación permite modificaciones, a veces es necesario hacer referencia al valor de una varialbe en el pre-estado (estado previo a la ejecución de la función). Se hace mediante la sintaxis **pre(*e*)**.

aux *definiciones*;

Las *definiciones* son de la forma ***F*(*parámetros*): *tipo* = *e***, separadas por comas. Cada una define un nombre de función auxiliar (*F*) que puede ser usado en el resto de la especificación para reemplazar a la expresión *e*. Los *parámetros* (opcionales, con la sintaxis ya vista) pueden aparecer libres en *e* y se reemplazan por las expresiones correspondientes cada vez que se utiliza *F*. Los parámetros del problema especificado también pueden aparecer en *e*. En *e* no se puede mencionar *F*. El nombre puede ser usado en el bloque donde aparece o en toda la especificación si no está dentro de un bloque. El *tipo* puede omitirse, porque se deduce del de la expresión, pero es buena práctica incluirlo.

Tipos Compuestos

Cada valor de un tipo básico representa un elemento atómico, indivisible. En cambio, un valor de un tipo compuesto contiene información que puede ser dividida en componentes de otros tipos. Ya vimos dos ejemplos de tipos compuestos: las secuencias y las tuplas. Un valor de tipo secuencia de enteros tiene varios componentes, cada uno es un entero. Un valor de tipo par ordenado de enteros tiene dos componentes.

Para definir un tipo compuesto, tenemos que darle un nombre y uno o más **observadores**. Los observadores son funciones que se aplican a valores del tipo compuesto (pueden tener más parámetros) y devuelven el valor de sus componentes. Se usan en el resto de la especificación (precondiciones, poscondiciones, funciones auxiliares).

Los observadores son los que definen el tipo compuesto. Si dos términos del tipo dan el mismo resultado para todos los observadores, se los considera iguales (esta es la definición implícita de la igualdad, ==, para los tipos compuestos).

Si un término del tipo tiene valor definido, entonces ninguna operación (ni siquiera los observadores) puede dar resultado indefinido al aplicársele. Para garantizarlo, los observadores pueden tener precondiciones. Por el contrario, los observadores no tienen poscondiciones. Si hay condiciones generales que deban cumplir los elementos del tipo, se las presenta como **invariantes de tipo**.

El mecanismo más habitual para especificar funciones que tienen como resultado elementos de un tipo compuesto es indicando cuánto valen los observadores aplicados al resultado.

Ejemplos

Veamos algunos ejemplos de tipos compuestos:

- El tipo `Punto` representa un punto en el plano. Sus componentes son las dos coordenadas (`X` e `Y`); tiene un observador para obtener cada una de ellas.

```
tipo Punto {
   observador X(p: Punto): Float;
   observador Y(p: Punto): Float;
}
```

  Especifiquemos ahora una problema que reciba dos números reales y construya un punto con esas coordenadas:

```
problema nuevoPunto(a, b: Float) = result: Punto {
   asegura X(result) == a
   asegura Y(result) == b;
}
```

  En este ejemplo se ve cómo especificar un problema que tiene un tipo compuesto como resultado: indicando qué se obtiene al aplicarle cada uno de los observadores.

  Si queremos definir una función auxiliar para calcular la distancia entre dos puntos, no vamos a usar esta misma técnica; porque el resultado no es de tipo compuesto. Pero sí vamos a aplicar los observadores a los parámetros, para obtener los componentes de los argumentos:

```
aux dist(p, q: Punto): Float = ((X(p)-X(q))^2+(Y(p)-Y(q))^2)^1/2;
```

- Ahora vamos a aprovechar el tipo compuesto que acabamos de definir para definir un tipo nuevo:

```
aux tri(a, b, c: Punto): Bool = dist(a,b) < dist(a,c) + dist(b,c);
```

```
tipo Triángulo {
   observador V1(t: Triángulo): Punto;
   observador V2(t: Triángulo): Punto;
   observador V3(t: Triángulo): Punto;
   invariante tri(V1(t),V2(t),V3(t));
   invariante tri(V1(t),V3(t),V2(t));
   invariante tri(V2(t),V3(t),V1(t));
}
```

Los invariantes de tipo son la garantía de que los elementos estén bien construidos, deben cumplirse para cualquier elemento del tipo. En este caso, se piden las desigualdades triangulares (representadas por la función auxiliar `tri`) para que no haya un par de puntos iguales y los puntos no estén alineados.

Todos los problemas que reciban valores del tipo compuesto como argumentos pueden suponer, además de su precondición, que se cumplen los invariantes. Y los que calculen resultados de ese tipo tienen que asegurar que los invariantes no se van a violar.

Por ejemplo, especifiquemos el problema de construir un triángulo a partir de tres puntos, garantizando las desigualdades triangulares:

```
problema nuevoTriángulo(a, b, c: Punto) = res: Triángulo {
   requiere tri(a,b,c) && tri(a,c,b) && tri(b,c,a);
   asegura V1(res) == a && V2(res) == b && V3(res) == c;
}
```

¿Qué pasaría si se hubiera omitido la precondición? Estaríamos especificando el problema de construir un triángulo a partir de tres puntos cualesquiera, sin aclarar en la poscondición qué se debe hacer cuando no cumplen la desigualdad triangular. Una función que resolviera el problema, generaría triángulos que no respetarían el invariante del tipo, lo cual no es aceptable. La especificación es errónea porque no admite ninguna solución. Todos los problemas que especifiquemos deben garantizar el cumplimiento de los invariantes de tipo.

- El típo Círculo es parecido al que acabamos de definir, pero difiere en que sus componentes son de tipos distintos.

```
tipo Círculo {
   observador Centro(c: Círculo): Punto;
   observador Radio(c: Círculo): Float;
   invariante Radio(c) > 0;
}
```

Los dos problemas siguientes especifican funciones que construyen un círculo. Una, a partir de su centro y su radio; la otra, a partir de su centro y un punto sobre la circunferencia:

```
problema nuevoCírculo(c: Punto, r: Float) = circ: Círculo {
   requiere r > 0;
   asegura Centro(circ) == c && Radio(circ) == r;
}

problema nuevoCírculoPuntos(c, x: Punto) = circ: Círculo {
   requiere dist(c,x) > 0;
   asegura Centro(circ) == c && Radio(circ) == dist(c,x);
}
```

Tipos genéricos

En los ejemplos de tipos compuestos que vimos recién, cada uno de los componentes es de un tipo predeterminado. Hay otros tipos compuestos que representan **estructuras** cuyo contenido van a ser valores, no siempre del mismo tipo. Un ejemplo que ya hemos visto son las secuencias. Cada secuencia representa una colección ordenada de elementos que pueden ser de cualquier tipo (aunque todos deben ser del mismo). Las secuencias estructuran elementos de un tipo siempre de la misma forma, sin importar de qué tipo se trate. Existen secuencias de enteros, de reales, de círculos y de todos los tipos que definamos. Incluso secuencias de secuencias de enteros. Y todas las secuencias tienen el mismo comportamiento: las especificaciones y definiciones de funciones básicas valen para secuencias de todo tipo.

Estos tipos se llaman **tipos genéricos** o **tipos paramétricos**, porque el nombre del tipo tiene parámetros, variables que representan a los tipos de los componentes. Los parámetros de tipo se escriben entre los símbolos `<` y `>` (o ⟨ y ⟩) después del nombre del tipo. Por ejemplo, `Matriz<T>` es el tipo genérico de las matrices cuyos elementos pertenecen a un tipo al que, en la especificación de las matrices, nos vamos a referir como `T`.

Vimos que los problemas tenían distintas instancias, una para cada combinación de valores de sus parámetros. Lo mismo ocurre con los tipos genéricos. Por ejemplo, una **instancia del tipo genérico** `Matriz<T>` es `Matriz<Char>`, el tipo de las matrices cuyos elementos son caracteres. Un tipo genérico define entonces, en realidad, una familia de tipos; cada elemento de la familia es una instancia del tipo genérico. Por ejemplo, `Matriz<Char>` es uno de los miembros de la familia definida por `Matriz<T>`; otros miembros son `Matriz<Int>`, `Matriz<Día>`, `Matriz<Triángulo>`.

Ejemplos

Veamos algunos ejemplos de tipos compuestos genéricos:

- Empezamos por el que mencionamos en la explicación, las matrices. Las operaciones que definen una matriz son sus dimensiones (número de filas y columnas) y la obtención del contenido en una posición:

```
tipo Matriz<T> {
   observador filas(m: Matriz<T>): Int;
   observador columnas(m: Matriz<T>): Int;
   observador val(m: Matriz<T>, f, c: Int): T {
      requiere 0 <= f < filas(m);
      requiere 0 <= c < columnas(m);
   }
   invariante filas(m) > 0;
   invariante columnas(m) > 0;
}
```

  También podemos definir y especificar **operaciones genéricas**, funciones cuya descripción dependa únicamente de la estructura del tipo genérico y, por lo tanto, no necesiten definirse sobre una instancia en particular. Un ejemplo es la función `elementos`, que cuenta la cantidad de elementos de una matriz:

```
aux elementos(m: Matriz<T>): Int = filas(m) * columnas(m);
```

  Especifiquemos ahora el problema de cambiar el valor de una posición de una matriz:

```
problema cambiarM(m: Matriz<T>, f, c: Int, v: T) {
   requiere 0 <= f < filas(m);
   requiere 0 <= c < columnas(m);
   modifica m;
   asegura val(m,f,c) == v;
   asegura (∀ i<-[0.. filas(m)), j<-[0.. columnas(m)),i≠f || j≠c)
         val(m,i,j) == val(pre(m),i,j);
}
```

En la poscondición hizo falta asegurar que los valores de las demás posiciones de la matriz no se modificaban, porque la cláusula `modifica` solo avisa que `m` puede cambiar, pero no qué posiciones van a cambiar (recordar que en la cláusula modifica sólo se pueden listar variables). Es por esto que en la poscondición tenemos que escribir explícitamente que las posiciones distintas a `(f,c)` no cambian de valor.

Y, por supuesto, podemos también especificar problemas para instancias particulares del tipo genérico. Por ejemplo, el que construye la matriz identidad de $n \times n$:

```
problema matId(n: Int) = result: Matriz<Int> {
   requiere n > 0;
   asegura filas(result) == columnas(result) == n;
   asegura (∀ i<-[0..n)) val(result,i,i) == 1;
   asegura (∀ i<-[0..n), j<-[0..n), i≠j) val(result,i,j) == 0;
}
```

- Pasemos ahora al tipo genérico `Secuencia<T>`. Lo usamos desde que empezamos a ver el lenguaje de especificación, con su nombre alternativo: `[T]`. Ya presentamos también los observadores que usamos para el tipo: la longitud y la indexación:

```
tipo Secuencia<T> {
   observador long(s: Secuencia<T>): Int;
   observador índice(s: Secuencia<T>, i: Int): T {
      requiere 0 <= i < long(s);
   }
}
```

Recordemos las notaciones alternativas: $|s|$ para la longitud y s_i o $s[i]$ para la indexación.

Si bien las secuencias son un tipo genérico compuesto como otros, tienen un lugar especial en nuestro lenguaje de especificación. Este privilegio se lo dan las notaciones para construir secuencias por comprensión e intervalos, que no son compartidas por otros tipos.

La siguiente función auxiliar sirve para ver si una secuencia es subsecuencia contigua de otra:

```
aux ssc(a,b: [T]): Bool = (∃ i <- [0..|b|-|a|]) a == b[i..(i+|a|));
```

Y ahora que sabemos cuáles son los observadores, podemos definir algunas de las otras funciones auxiliares que presentamos para las secuencias:

```
aux cab(a: [T]): T = a[0];
aux cola(a: [T]): [T] = a[1..];
aux en(t: T, a: [T]): Bool = [x | x <- a, x == t] ≠ [];
aux sub(a: [T], d, h: Int): [T] = [a[i] | i ∈ [d..h]];
aux todos(sec: [Bool]): Bool = False ∉ sec;
aux alguno(sec: [Bool]): Bool = True ∈ sec;
```

Para algunas de estas operaciones no usamos directamente los observadores, aprovechamos la notación de listas por comprensión. Es importante notar que el selector `<-` es parte de la notación de listas por comprensión y no una aplicación de la función `en` (aunque en nuestra notación ambos pueden reemplazarse por el símbolo ∈).

Para representar textos (cadenas de caracteres) vamos a usar el tipo `Secuencia<Char>` (o `[Char]`). Supongamos que tenemos una lista de palabras y queremos ver si alguna de ellas aparece en un libro. Esta sería una forma de especificar el problema:

```
problema hayAlguna(palabras: [[Char]], libro: [Char])= res: Bool {
   requiere NoVacías: (∀ p <- palabras ) |p| > 0;
   requiere SinEspacios: ¬(∃ p <- palabras) ' ' en p;
   asegura res == (∃ p <- palabras) ssc(p,libro);
}
```

Para la lista de palabras usamos el tipo `[[Char]]`, que también se puede escribir `Secuencia<Secuencia<Char>>` y tiene como valores secuencias que en cada posición tienen una secuencia de caracteres.

Como sus nombres lo indican, las dos precondiciones limitan el uso de la función a los casos en que cada palabra tenga al menos una letra y no contenga ningún espacio (dejaría de ser una palabra). También podríamos haber reemplazado la precondición `SinEspacios` por una que pidiera que las palabras no contuvieran ningún símbolo que no fuera una letra, de la siguiente manera:

```
requiere SinSímbolos: (∀ p <- palabras) soloLetras(p);
aux letras: [Char] = ['A'..'Z'] ++ ['a'..'z'];
aux soloLetras(s: [Char]): Bool = (∀ l <- s) l en letras;
```

- Otros tipos genéricos que ya mencionamos son las tuplas. Se trata de secuencias de tamaño fijo, cada uno de cuyos elementos puede pertenecer a un tipo distinto. Como ejemplo, veamos los pares y las ternas:

```
tipo Par<A,B> {
   observador prm(p: Par<A,B>): A;
   observador sgd(p: Par<A,B>): B;
}
tipo Terna<A,B,C> {
   observador prm3(t: Terna<A,B,C>): A;
   observador sgd3(t: Terna<A,B,C>): B;
   observador trc3(t: Terna<A,B,C>): C;
}
```

Ya habíamos visto que `Par<A,B>` también se podía escribir `(A,B)`; y `Terna<A,B,C>` es `(A,B,C)`.

La siguiente especificación describe una función (inútil) que toma un par formado por un entero y un carácter, y devuelve otro par del mismo tipo, con el doble del entero y el siguiente carácter:

```
problema otroPar(p: (Int,Char)) = result: (Int,Char) {
   asegura prm(result) == 2*prm(p);
   asegura sgd(result) == char(ord(sgd(p))+1);
}
```

¿Cuál es la diferencia entre escribir esta especificación y definir una función auxiliar que devuelva un par con estas mismas características? Al haberlo planteado como un problema, nos estamos proponiendo construir una función (en un lenguaje de programación) que cumpla la especificación. Si hubiéramos definido una función auxiliar, podríamos usarla en el resto de la especificación.

Para construir una tupla, se escriben los términos correspondientes a cada componente como una lista separada por comas y encerrada entre paréntesis. Por ejemplo, las poscondiciones de la función anterior podrían haberse escrito así:

```
asegura result == (2*prm(p), char(ord(sgd(p))+1));
```

Sintaxis

La definición de un tipo compuesto tiene esta forma:

tipo *nombre<parTipo> declaración*

parTipo es una lista opcional de variables de tipo separadas por coma. Las variables aparecen en la *declaración* representando tipos de datos.

La *declaración* son sentencias de tipo entre llaves, encabezadas por una palabra clave (`observador`, `invariante`, `aux`):

observador *nombre*(*parámetros*)**:** *tipoRes contrato*

Los *parámetros* son de la forma conocida y el *contrato* no incluye poscondiciones (cláusulas `asegura`). No se le pone nombre al resultado porque, al no haber poscondición, no es necesario referenciarlo.

invariante *nombre*: *P*;

El *nombre* es opcional. *P* es un predicado que puede tener una única variable libre, la cual se interpreta como "cualquier valor del tipo especificado". Esta expresión debe ser válida antes y después de la invocación a todas las funciones, por lo que cualquier especificación o definición de función puede suponerla válida (sin necesidad de tenerlo como parte de su precondición). Los problemas tienen la obligación de cumplirla en su poscondición (sin que figure explícitamente).

aux *definiciones*;

Es la misma sentencia que hemos visto para las especificaciones. Cuando figura dentro de una declaración de un tipo, los nombres definidos pueden usarse en toda la declaración. Si se quiere que estén disponibles en la especificación de otras funciones, la sentencia debe colocarse fuera de la declaración.

Función ifThenElse

La función `IfThenElse` elige entre dos elementos del mismo tipo, de acuerdo a una *guarda*. Si la guarda es verdadera, elige el primero de los dos; si no, elige el segundo. Por ejemplo, `ifThenElse(x!=0, 1/x, 8)` es el valor de `1/x` cuando `x` es distinto de

cero y es 9 en caso contrario. También se puede escribir if x!=0 Then 1/x Else 8. Observar que IfThenElse es una función no estricta.

Con esta función podemos definir también algunas otras de las funciones auxiliares que estamos usando para las secuencias:

aux cons(t: T, a: [T]): [T] = [if i==-1 then t else a[i] | i ∈ [-1..|a|)];

aux conc(a, b: [T]): [T] = [if i<|a| then a_i else $b_{i-|a|}$ | i ∈ [0..|a|+|b|)];

aux cambiar(a: [T], i: Int, v: T): [T] = [if i≠j then a_i else v | j ∈ [0..|a|)];