Capítulo 10

El Proceso de construcción de programas

Índice del Capítulo

10.1. Introducción

El desarrollo de software es un proceso por el cual, dado un problema, se encuentra un programa (o un conjunto de programas) que lo resuelva eficientemente.

Una de las dificultades esenciales de este proceso consiste en que la descripción del problema a resolver, suele ser poco precisa o incompleta. Sin embargo, esta dificultad no cobró importancia hasta el final de la década del 60', dado que hasta ese momento las computadoras sólo eran usadas para resolver problemas científicos, los cuales estaban expresados en un lenguaje suficientemente preciso.

Con el abaratamiento de los costos las computadoras comenzaron a ser utilizadas para problemas originados en otros ámbitos (en general problemas administrativos). El principal método utilizado entonces, era simplemente partir del problema, el cual estaba expresado de manera informal y poco detallada, y obtener un programa que, por definición, es muy detallado y está escrito en una notación formal. Este salto, fue una de las principales razones de la llamada "crisis del software", la cual produjo un decaimiento en la confiabilidad del software y por lo tanto un decaimiento de los métodos usados para su desarrollo. Otra dificultad de esta metodología, se presentaba a la hora de la corrección del programa. Para ello, se realizaban ensayos con conjuntos de datos para los cuales se conocía el resultado, y si éstos daban los resultados esperados se daba por terminada la tarea. En el caso frecuente en que los resultados no fueran

correctos, se procedía a modificar el programa para intentar corregirlo. Esta tarea era sumamente difícil, debido a que no se sabía a priori si el resultado inesperado se debía a meros errores de programación o a una concepción inadecuada del problema.

En la actualidad, es ampliamente aceptado que el proceso de construcción de programas, debe dividirse en al menos dos etapas: la etapa de *especificación* del problema y la etapa de *programación* o desarrollo del programa.

El resultado de la primera etapa es una *especificación formal* del problema, la cual seguirá siendo abstracta (poco detallada) pero estará escrita con precisión en algún lenguaje cuya semántica esté definida rigurosamente. La segunda etapa dará como resultado un programa y una demostración de que el programa es *correcto* respecto de la especificación dada.

La separación en dos etapas, permite discernir ahora si un programa cuyos resultados no son los esperados, es incorrecto o si, en cambio, es la especificación la que no describe al programa de la manera adecuada.

Este paradigma de desarrollo de software, (primero escribir una especificación clara y luego desarrollar una implementación aceptablemente eficiente), ha sido un foco de investigación activa durante los últimos veinte años, y no debería tomarse como un método panacea aplicable a todas las circunstancias. Sin embargo, se puede mejorar mucho la fiabilidad de los programas tratando de aplicar estos principios siempre que sea posible.

En lo que sigue de este capítulo, ilustraremos a través de ejemplos como pueden construirse especificaciones para problemas relativamente simples (pero para nada triviales).

10.2. Especificaciones

Podemos decir, en un sentido general que una especificación es una descripción formal de la tarea que un programa tiene que realizar. Es común, también definir a la especificación como la respuesta a la pregunta ¿qué hace el programa?, mientras que el programa mismo, conocido como *implementación*, responde a la pregunta ¿cómo se realiza la tarea?

En la literatura, se presenta también a una especificación como un *contrato* entre el programador y el potencial usuario. Este contrato establece de manera precisa cual es el comportamiento del producto que el programador debe proveer al usuario. Usualmente las especificaciones dicen que si el programa se ejecuta para un valor de un conjunto dado, el resultado va a satisfacer una cierta propiedad. Una de las consecuencias del contrato, es que el usuario se compromete a usar el programa sólo para valores de ese conjunto predeterminado y el programador a asegurar que el programa producirá un resultado satisfactorio. En principio, si el usuario ingresa al programa datos que no están en el conjunto de datos aceptables, el comportamiento del programa puede ser cualquiera sin que ello viole el contrato establecido por la especificación.

En el contexto de nuestro formalismo básico, un modo de especificar una función consiste en establecer explícitamente la regla de correspondencia entre argumentos y resultados. La notación que hemos utilizado puede ser muy expresiva y, a veces, una especificación de una función es de hecho un programa. En este caso, se puede ejecutar la especificación directamente. Sin embargo, puede ser tan altamente ineficiente que la posibilidad de ser ejecutada será simplemente de interés teórico. A pesar de haber escrito una especificación ejecutable, el programador

no queda necesariamente eximido de la tarea de producir una versión alternativa equivalente, pero aceptablemente eficiente.

Tomemos por ejemplo, el siguiente problema: dado el conjunto de alumnos de un curso, se pide obtener el alumno con el segundo promedio más alto. Este problema tiene una formulación bastante precisa, pero sin embargo existen ciertas cuestiones sin resolver. Por ejemplo, puede ser que tal alumno no exista (es el caso en que todos los alumnos del curso tengan el mismo promedio), o que haya más de un alumno que cumpla con la misma propiedad. Si nos quedamos con un enunciado informal, la solución para estos casos queda al criterio del programador.

En lo que queda del capítulo, usaremos herramientas introducidas anteriormente para escribir especificaciones precisas. Para ello, interpretaremos las fórmulas del cálculo de predicados y las expresiones cuantificadas como es usual y construiremos con ellas y con el formalismo básico las especificaciones formales. Esto nos permitirá resolver las ambigüedades en la formulación de problemas.

10.3. Ejemplos

(10.1) **Ejemplo.** En el siguiente ejemplo, el usuario propone al programador el problema de calcular la raíz cuadrada de un número real dado.

Primer intento El programador que cuenta con conocimientos de matemática, propone resolver el problema sólo para números reales que no sean negativos. El usuario acepta la propuesta, con lo cual el contrato entre ambos (la especificación) queda establecido como sigue:

El programador se compromete a construir una función $sqrt : Real \to Real$, tal que:

$$(\forall\, x : 0 \leq x : (sqrt.x)^2 = x)$$

Puede pensarse a la especificación como una ecuación a resolver donde la incógnita es *sqrt*, es decir, el problema es encontrar una función *sqrt* de modo que satisfaga la especificación. Nótese que esta función no necesariamente es única, dado que hay dos soluciones posibles para la raíz cuadrada de un número estrictamente positivo. Esto da lugar a que haya infinitas soluciones, dado que en principio, pueden elegirse determinados valores para los cuales la raíz debe ser positiva y otros para los cuales debe ser negativa.

Una forma estilizada de escribir la misma especificación es separar la **precondición** de la **postcondición**. La primera condición establece las restricciones que determinan los datos aceptables, mientras que la segunda establece las propiedades que serán satisfechas por el resultado. En nuestro ejemplo, podemos escribir:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & (sqrt.x)^2 = x \end{array}$$

Nótese que la cuantificación de la variable queda implícita. Esto será una práctica usual. En general se supondrá que las variables que aparecen como argumentos de la función especificada están cuantificadas universalmente, siendo esta cuantificación acotada al conjunto de valores definidos por el tipo de la función. La especificación original puede entonces escribirse simplemente así:

$$0 \leq x \Rightarrow (sqrt.x)^2 = x$$

Segundo intento Un fenómeno usual ocurre cuando el programador por alguna razón no puede satisfacer la especificación acordada. Por ejemplo, no será posible para el programador encontrar una solución exacta de $sqrt.2$, debido a que cualquier sistema de cómputo en el cual implemente su programa, sólo manejará aproximaciones finitas a los números irracionales. El problema aquí, es que la especificación requiere que la solución sea exacta, por lo tanto el programador tiene que cambiar el contrato con el usuario. Una posible especificación es:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & \left|(sqrt.x)^2 - x\right| < \epsilon \end{array}$$

donde ϵ es una constante a ser negociada.

Escrita de esta manera es obvio que la especificación es más favorable para el programador, pues la postcondición es más débil y por lo tanto más fácil de satisfacer. Si escribimos esta nueva especificación:

$$0 \leq x \Rightarrow \left|(sqrt.x)^2 - x\right| < \epsilon$$

es claro que la especificación del primer intento implica ésta.

Cambios en la especificación:

Como se vio en el ejemplo anterior, es frecuente la necesidad de modificar las especificaciones. En el ejemplo, la especificación fue *debilitada*, es decir se optó por una especificación que requería menos del programa. La forma de debilitarla fue a través de la postcondición. Otra forma de debilitar la especificación es fortalecer la precondición, es decir exigiendo que el programa pueda funcionar sólo con un conjunto más restringido de valores. Por ejemplo, podríamos exigir como requisito que los argumentos para *sqrt* sean cuadrados perfectos (con lo cual siempre será posible encontrar valores exactos para las raíces). La especificación entonces, resulta:

$$\begin{array}{ll} pre: & (\exists y : y \in \mathbb{N} : y^2 = x) \\ post: & (sqrt.x)^2 = x \end{array}$$

Otra situación común es la necesidad de fortalecer la especificación. Esto ocurre cuando el usuario observa que necesita un programa con mejores propiedades que el que tiene, o también

cuando es necesario utilizar el programa para un conjunto de datos más amplio del que consideraba inicialmente. En el ejemplo anterior, puede requerirse que el resultado sea siempre no negativo, con lo cual se está fortaleciendo la postcondición y la especificación completa:

$$0 \leq x \Rightarrow \left|(sqrt.x)^2 - x\right| < \epsilon \wedge 0 \leq sqrt.x$$

(10.2) Ejemplo.

Consideremos ahora la especificación de una función que calcula la segunda mejor nota de un curso. Supondremos que las notas de los alumnos están almacenadas en una lista, con lo cual el tipo de la función sería $[Nat] \rightarrow Nat$. Una manera simple de construir esta especificación es usando una especificación auxiliar que exprese la mayor nota del curso:

$$\begin{array}{ll} pre: & true \\ post: & maxNota.xs = (max\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

La especificación de la función requerida es entonces:

$$\begin{array}{ll} pre: & (\exists\, i, j : 0 \leq i < j < \#xs : \neg(xs.i = xs.j)) \\ post: & segNota.xs = (max\, i : 0 \leq i < \#xs \wedge \neg(xs.i = maxNota.xs) : xs.i) \end{array}$$

La función *maxNota* es una función auxiliar usada para especificar la función *segNota*, y no es en principio necesario construir un programa para ella. Durante el proceso de construcción del programa para *segNota* se verá si es también necesario construir un programa para la función auxiliar.

No deben confundirse las igualdades que aparecen en una especificación, denotadas con el signo " = ", con las definiciones del formalismo básico, para las cuales definimos y utilizamos el signo " $\doteq$ ".

Notemos que la precondición de esta función, determina que la función no está definida para listas que no contienen al menos dos elementos distintos. Es necesaria esta restricción para estar seguros de que la segunda mejor nota de un curso existe y podamos calcularla.

10.4. Más ejemplos

Dado que no existen reglas generales para determinar las especificaciones de problemas, incluímos a continuación algunos ejemplos. La clave está en saber expresar los problemas planteados en lenguaje corriente de manera concisa, usando el lenguaje formal.

Escribiremos el problema entre comillas. Cabe aclarar que con frecuencia, los problemas pueden especificarse de diferentes formas, con lo cual no deben tomarse las especificaciones que aparezcan a continuación como las únicas posibles.

(10.3) Ejemplo. "Dados $x \geq 0$ e $y > 0$ enteros, calcular el cociente y el resto de la división entera de x por y".

El algoritmo de la división entera nos asegura la existencia de un único entero q cociente de la división entera de x por y y un único entero r resto de esta división. Estos números están caracterizados por las propiedades $x = q \times y + r$ y $0 \leq r < y$. Por lo tanto, una función $divMod : Int \to Int \to (Int, Int)$, que calcule la división entera y el resto de la división entera, puede especificarse como:

$$\begin{array}{ll} pre: & x \geq 0 \ \wedge \ y > 0 \\ post: & divMod.x.y = (q, r) \equiv x = q \times y + r \ \wedge \ 0 \leq r < y \end{array}$$

(10.4) Ejemplo. "Dado un número entero $x \geq 0$, $sqrtInt.x$ es la raíz cuadrada de x".

La raíz cuadrada entera de un número está definida como el mayor entero positivo cuyo cuadrado es menor o igual que el número en cuestión. La función $sqrtInt : Int \to Int$ puede expresarse en lenguaje formal de la siguiente manera:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & sqrtInt.x = (max\, z : 0 \leq z \wedge z^2 \leq x) \end{array}$$

alternativamente, puede evitarse el cuantificador máximo de la siguiente manera:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & 0 \leq sqrtInt.x \ \wedge \ (sqrInt.x)^2 \leq x \ \wedge \ x < (sqrtInt.x + 1)^2 \end{array}$$

(10.5) Ejemplo.

Sea xs una lista no vacía de enteros, "$m.xs$ es el mínimo de xs".

Usando el cuantificador min, la especificación de $m : [Int] \to Int$ es inmediata:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & m.xs = (min\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

Otra especificación posible se obtiene describiendo con más detalles el mínimo:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & (\forall\, i : 0 \leq i < \#xs : xs.i \geq m.xs) \wedge (\exists\, i : 0 \leq i < \#xs : xs.i = m.xs) \end{array}$$

(10.6) Ejemplo.

Sea xs una lista no vacía de enteros, "$iguales.xs$ determina si todos los elementos son iguales".

Una manera de especificar la función $iguales : [Int] \to Bool$ se obtiene al considerar que todos los elementos de la lista son iguales si cada elemento es igual al de su derecha.

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & iguales.xs = (\forall\, i : 0 \leq i < \#xs - 1 : xs.i = xs.(i+1)) \end{array}$$

Pero también podríamos expresar que todos son iguales diciendo que cada uno de ellos es igual a uno fijo, por ejemplo, el primero (sabemos que existe, pues suponemos que la lista es no vacía).

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & iguales.xs = (\forall\, i : 0 < i < \#xs : xs.i = xs.0) \end{array}$$

(10.7) Ejemplo.

"La función $creciente : [Int] \to Bool$ toma una lista no vacía de enteros y determina si la lista está ordenada en forma creciente.".

Si una lista está ordenada en forma creciente, cada elemento de ésta debe ser mayor que el anterior.

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & creciente.xs = (\forall\, i : 0 < i < \#xs : xs.i > xs.(i-1)) \end{array}$$

Observemos que fue necesario excluir al cero del rango de especificación, para que el término esté siempre definido.

También puede especificarse esta función expresando la relación de desigualdad correspondiente entre cualquier par de índices:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & creciente.xs = (\forall\, i : 0 \leq i \leq \#xs : (\forall\, j : i < j < \#xs : xs.i < xs.j)) \end{array}$$

(10.8) Ejemplo. "La función $f : Int \to [Int] \to Bool$ determina si el k-ésimo elemento de la lista xs aloja el mínimo valor de xs".

Notemos que la función f debe aplicarse a dos argumentos, un entero que indica una posición de la lista y una lista. Para que la evaluación tenga sentido, el entero debe estar comprendido en el rango que corresponde al tamaño de la lista. Entonces, podemos escribir:

$$\begin{array}{ll} pre: & 0 \leq k < \#xs \\ post: & f.k.xs = (xs.k = (min\, i : 0 \leq i < \#xs : xs.i)) \end{array}$$

o bien

$$\begin{array}{ll} pre: & 0 \leq k < \#xs \\ post: & f.k.xs = (\forall\, i : 0 \leq i < \#xs : xs.k \leq xs.i) \end{array}$$

(10.9) Ejemplo. "La función $minout : [Nat] \to Nat$ selecciona el menor natural que no está en xs".

La función *minout* extrae el mínimo de un conjunto, éste deberá especificarse dentro del rango, por ejemplo:

$$\begin{array}{ll} pre: & true \\ post: & minout.xs = (min\, i : 0 \leq i \wedge (\forall\, j : 0 \leq j < \#xs : xs.j \neq i) : i) \end{array}$$

(10.10) Ejemplo. "La función *lmax* toma una lista xs de enteros y calcula la longitud del máximo intervalo de la forma $[xs.0, \ldots, xs.k]$, que verifica $xs.i = 0$ para todo $0 \leq i < k$".

Analicemos primero la función *lmax*. Si el primer elemento de la lista xs no es igual a 0, entonces no existe un intervalo de la forma $[xs.0, \ldots, xs.k]$ con elementos nulos, con lo cual la función debe devolver 0. En caso contrario, la lista xs contiene un intervalo nulo y debemos calcular su longitud. Una especificación posible para la función *lmax* es la siguiente:

$$\begin{array}{ll} pre: & true \\ post: & lmax.xs = max(0, (max\, k : 0 \leq k < \#xs \wedge (\forall\, i : 0 \leq i < k : xs.i = 0) : k)) \end{array}$$

Esto también puede escribirse usando listas en lugar del entero k:

$$\begin{array}{ll} pre: & true \\ post: & lmax.xs = max(0, (max\, as, bs : xs = as +\!\!+ bs \wedge \\ & (\forall\, i : 0 \leq i < \#as : xs.i = 0) : \#as)) \end{array}$$

(10.11) Ejemplo. "Dada una lista de enteros, la función P determina si algún elemento de la lista es igual a la suma de todos los anteriores a él".

La función $P : [Int] \to Bool$ se puede especificar como:

$$\begin{array}{ll} pre: & true \\ post: & P.xs = (\exists\, i : 0 < i < \#xs : xs.i = (\sum\, j : 0 \leq j < i : xs.j)) \end{array}$$

(10.12) Ejemplo. "Dado un polinomio $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, la función *ev* realiza la evaluación del polinomio para un valor determinado de la variable x. Para especificar esta función, debemos elegir una forma adecuada de representar el polinomio. Como un polinomio queda unívocamente determinado por sus coeficientes, una posibilidad es formar con éstos una lista: $xs = [a_0, a_1, \ldots, a_{n-1}, a_n]$. Más aun, cada lista de reales puede pensarse como lista de coeficientes de un polinomio. El polinomio nulo se representará con la lista [0] o con una lista cuyos elementos sean todos nulos, con esta convención la lista vacía no representa a ningún polinomio. Podemos especificar $ev : [Real] \to Real \to Real$ de la siguiente manera:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & ev.xs.y = (\sum\, i : 0 \leq i < \#xs : xs.i \times y^i) \end{array}$$

(10.13) Ejemplo. "Dada una lista de valores booleanos, la función $bal : [Bool] \to Bool$ indica si en la lista hay igual cantidad de elementos $True$ que $False$".

El cuantificador N nos permite contar la cantidad de $True$ y $False$ que hay en la lista. Lo que nos interesa es el resultado de la comparación de estas dos cantidades:

$$\begin{array}{ll} pre: & true \\ post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i) = \\ & (N\, i : 0 \leq i < \#xs : \neg xs.i)) \end{array}$$

o equivalentemente:

$$\begin{array}{ll} pre: & true \\ post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i) = (N\, i : 0 \leq i < \#xs : \neg xs.i)) \end{array}$$

(10.14) Ejemplo. "Dada una lista xs de booleanos, la función $balmax.xs$ indica la longitud del máximo segmento de xs que satisface bal".

Dada una lista xs, un segmento de ella es una lista cuyos elementos están en xs, en el mismo orden y consecutivamente. Por ejemplo, si $xs = [2, 4, 6, 8]$ entonces $[2, 4]$, $[4, 6, 8]$, $[\,]$, son segmentos, mientras que $[4, 2]$ y $[2, 6, 8]$ no lo son.

En problemas como éste, suele ser conveniente expresar la lista como una concatenación. Si escribimos $xs = as +\!\!+ bs +\!\!+ cs$, entonces as, bs y cs son segmentos de xs, por lo que podemos usarlos para expresar las condiciones que necesitamos que se satisfagan. Observemos que de los tres segmentos mencionados, el más general es bs, puesto que as comienza en la posición cero de xs, mientras que cs termina necesariamente en la posición última de xs. El segmento bs puede estar en cualquier parte, puesto que tanto as como cs pueden ser vacíos. Como queremos obtener la máxima longitud utilizamos el cuantificador max.

Una especificación de la función $balmax$ es la siguiente:

$$\begin{array}{ll} pre: & true \\ post: & balmax.xs = max(0, (max\, bs : (\exists\, as, cs :: xs = as +\!\!+ bs +\!\!+ cs) \wedge\ bal.bs : \\ & \#bs)) \end{array}$$

o también

$$\begin{array}{ll} pre: & true \\ post: & balmax.xs = max(0, (max\, as, bs, cs : xs = as +\!\!+ bs +\!\!+ cs\ \wedge\ bal.bs : \#bs)) \end{array}$$

Si la lista xs no contiene un segmento balanceado, $balmax$ debe devolver 0, por esta razón $balmax.xs$ es igual al máximo entre 0 y, el entero que corresponde a la longitud

del máximo segmento balanceado o $-\infty$ si xs no contiene un segmento balanceado. Esta especificación podría simplificarse si hubiésemos definido el operador *max* sobre naturales en lugar de enteros, dado que sobre los números naturales el elemento neutro de *max* es 0.

10.5. Ejercicios

1. Expresar en lenguaje formal los siguientes problemas. Para cada especificación enuncie: tipo de la función, pre, y post condiciones.

   i) Dado un entero $n > 0$: "$pot2.n$ es el mayor entero que vale a lo sumo n y es una potencia de 2".

   ii) Dado una lista xs de naturales: "Si $xs \neq [\,]$, $mayor.xs$ es el mayor elemento de xs. Si $xs = [\,]$, $mayor.xs$ es igual a 0".

2. Para cada uno de los problemas que se detallan a continuación consideraremos que la lista xs es no vacía y la misma contiene elementos enteros. Expresar en lenguaje formal cada una de las siguiente funciones, detallando el tipo, pre y post condición:

   i) Suponiendo que $\#xs > 1$ y que existen al menos dos valores distintos en xs: "$segundoMax.xs$ es el segundo valor más grande de xs".
   Ejemplo: si $xs = [3, 6, 1, 7, 6, 4, 7]$, debe resultar $segundoMax.xs = 6$.

   ii) "$sum.xs$ es la suma de los elementos de xs".

   iii) Dado un entero n: "El valor que describe $ocurrencias.n.xs$ corresponde a la cantidad de veces que n aparece en xs".

   iv) La función $distintos : [Int] \rightarrow Bool$ determina "si todos los valores de la lista que recibe como argumento son distintos".

   v) Dada una lista xs,"$pprod.xs$ denota el producto de todos los valores positivos de xs".

   vi) Dado un entero x: "$elem.x.xs$ coincide con el valor de verdad de la afirmación: x está en xs".

3. Sea xs una lista no vacía con elementos booleanos, tal que al menos un elemento de xs es $True$. Expresar en lenguaje formal los siguientes problemas indicando para cada uno ellos: el tipo de cada función que se especifica, pre y post condición.

   i) "$posmenor.xs$ es igual a la posición del primer elemento $True$ de la lista xs".
   Ejemplo: $posmenor.[true, false, false, true, true] = 0$.

   ii) "$posmayor.xs$ indica la posición del último elemento de la lista que es equivalente a $True$".
   Ejemplo: $posmayor.[true, false, false, true, true] = 4$

4. Sea xs una lista no vacía. Expresar las siguientes especificaciones en lenguaje corriente. Para ello tenga en cuenta que **sólo** podrán referenciarse en su redacción aquellas variables que aparecen libres en la especificación dada. No traduzca literalmente cada operador. Acompañe la redacción con dos ejemplos de aplicación de cada función.

   i) $f : \mathsf{Nat} \to [\mathsf{Int}] \to \mathsf{Bool}$
   $pre: \quad 0 < n \leq \#xs$
   $post: \quad f.n.xs = (\forall\, i : 0 \leq i < n : xs.i \geq 0)$

   ii) $h : \mathsf{Nat} \to [\mathsf{Int}] \to \mathsf{Bool}$
   $pre: \quad 0 < n \leq \#xs$
   $post: \quad h.n.xs = (\exists\, i : 0 \leq i < n : xs.i = 0)$

   iii) $g_1 : \mathsf{Nat} \to [\mathsf{Int}] \to \mathsf{Bool}$
   $pre: \quad 0 < n \leq \#xs$
   $post: \quad g_1.n.xs = (\exists\, i : \ 0 \leq i < n - 1 \ : xs.i < xs.(i+1) \vee xs.i > xs.(i+1))$

   iv) $g_2 : \mathsf{Nat} \to [\mathsf{Int}] \to \mathsf{Bool}$
   $pre: \quad 0 < n \leq \#xs$
   $post: \quad g_2.n.xs = (\forall\, i : \ 0 \leq i < n - 1 \ : xs.i < xs.(i+1) \vee xs.i > xs.(i+1))$

   v) $p : \mathsf{Nat} \to [\mathsf{Int}] \to \mathsf{Bool}$
   $pre: \quad 0 < n \leq \#xs$
   $post: \quad p.n.xs = (\forall\, i : 0 \leq i < n : par.(xs.i)) \wedge p.n.xs = g_2.n.xs$

5. Dadas las siguientes especificaciones con **múltiples postcondiciones** indique cuales de ellas describen formalmente el comportamiento pretendido para la función.

   a) "La función *maxl* calcula el máximo valor de una lista de enteros siempre y cuando la misma no sea vacía. En el caso que la lista fuera vacía, entonces la función *maxl* será igual al valor cero".

   $maxl : [\mathsf{Int}] \longrightarrow \mathsf{Int}$
   $Pre: \quad true$
   $Post_1: \quad maxl.xs = (\mathsf{max}\, i : 0 \leq i < \#xs : xs.i)$
   $Post_2: \quad maxl.xs = max(0, (\mathsf{max}\, i : 0 \leq i < \#xs : xs.i))$
   $Post_3: \quad (\#xs = 0 \Rightarrow maxl.xs = 0) \wedge$
   $\qquad (xs \neq [\,] \Rightarrow maxl.xs = (\mathsf{max}\, i : 0 \leq i < \#xs : xs.i))$
   $Post_4: \quad (\#xs = 0 \Rightarrow maxl.xs = 0) \vee$
   $\qquad (xs \neq [\,] \Rightarrow maxl.xs = (\mathsf{max}\, i : 0 \leq i < \#xs : xs.i))$

   b) "La función *minl* calcula el mínimo de una lista de enteros siempre y cuando la misma no sea vacía. En el caso que la lista fuera vacía, entonces la función *minl* será igual al valor cero".

$minl$: [Int] $\longrightarrow$ Int
Pre: $true$
$Post_1$: $minl.xs = min(0, (\min i : 0 \leq i < \#xs : xs.i))$
$Post_2$: $(\#xs = 0 \Rightarrow minl.[\,] = 0) \wedge$
$(\#xs > 0 \Rightarrow minl.xs = (\min i : 0 \leq i < \#xs : xs.i))$
$Post_3$: $minl.xs = k \;\Rightarrow\; (\forall i : 0 \leq i < \#xs : xs.i \geq k)$
$Post_4$: $minl.xs = k \;\equiv\; (\forall i : 0 \leq i < \#xs : xs.i \geq k) \wedge (\exists i : 0 \leq i < \#xs : xs.i = k)$
$Post_5$: $(\exists i : 0 \leq i < \#xs : xs.i = minl.xs) \wedge (\forall i : 0 \leq i < \#xs : xs.i \geq minl.xs)$

c) Dada xs una lista no vacía de enteros, la función *maxminl* calcula el máximo y el mínimo de la lista.

$maxminl$: [Int] $\longrightarrow$ (Int, Int)
Pre: $\#xs > 0$
$Post_1$: $maxminl.xs = (a, b) \equiv a = (\max i : 0 \leq i < \#xs : xs.i) \wedge$
$b = (\min i : 0 \leq i < \#xs : xs.i)$
$Post_2$: $maxminl.xs = ((\max i : 0 \leq i < \#xs : xs.i), (\min i : 0 \leq i < \#xs : xs.i))$
$Post_3$: $maxminl.xs = (maxl.xs, minl.xs)$
$Post_4$: $maxminl.xs.\,0 = maxl.xs \wedge maxminl.\,1 = minl.xs$

6. Sea xs una lista no vacía de enteros. Especificar las siguientes funciones detallando el tipo, la pre y post condición para cada caso.

i) La función $desc.xs$ determina si la lista xs está ordenada en forma descendiente.

> "La lista xs esta **ordenada en forma descendiente** si para todos par de elementos consecutivos de la misma, x_i e x_{i+1}, estos guardan entre si la relación de ser $x_i > x_{i+1}$".

ii) La función $noasc.xs$ determina si la lista xs está ordenada en forma no ascendente.

> "La lista xs esta **ordenada en forma no ascendente** si para todos par de elementos consecutivos de la misma, x_i e x_{i+1}, estos guardan entre si la relación de ser $x_i \geq x_{i+1}$".

iii) La función $promedio.xs$ calcula el promedio de todos los elementos de la lista.

7. Sea xs una lista no vacía con elementos reales, tal que al menos un elemento de xs es igual a cero. Especifique formalmente las siguientes funciones detallando su tipo, pre y post condiciones:

a) **acaCero.xs**: calcula una posición de la lista de xs donde ocurre un cero.

b) **prCero.xs**: calcula la posición de la lista de xs donde ocurre por primera vez el valor cero.

c) **úlCero.xs**: calcula la posición de la lista de xs donde ocurre por última vez el valor cero.

d) **segCero.xs**: calcula la posición de la lista de *xs* donde ocurre por segunda vez el valor cero.

e) **penCero.xs**: calcula la posición de la lista de *xs* donde ocurre por penúltima vez el valor cero.

8. Especificar las siguientes funciones. Detalle pre y post condición en cada una de ellas.

i) cp : [Int] → Nat

$cp.xs$ determina la cantidad de números pares que contiene xs.

ii) f : [Int] → Bool

$f.xs$ determina si xs contiene igual cantidad de elementos pares que impares.

iii) *noNulo* : [Int] → Bool

$noNulo.xs$ es $True$ si y sólo si xs no contiene elementos nulos.

iv) *prod* : Int → [Int] → Bool

$prod.x.xs$ es $True$ si y sólo si x es igual al producto de los elementos de xs.

v) g : Nat → [Int] → Bool

$g.k.xs$ determina si el k-ésimo elemento de xs aloja al máximo valor de xs.

vi) h : Nat → [Int] → Bool

$h.k.xs$ determina si el k-ésimo elemento de xs aloja la primera ocurrencia del máximo valor de xs.

vii) *meseta* : [Int] → Nat → Nat → Bool

$meseta.xs.i.j$ determina si todos los valores de la lista xs que están entre los índices i y j (ambos incluídos) son iguales.

viii) *ordenada* : [Int] → Nat → Nat → Bool

$ordenada.xs.i.j$ determina si la lista xs está ordenada entre los índices i y j (ambos incluídos). Notar que "ordenado" puede ser creciente o decreciente.

9. Traduzca al lenguaje coloquial las siguientes especificaciones. No traducir literalmente cada operador. Como sugerencia para esto último, confeccione inicialmente una gráfica genérica de una lista que cumpla con las condiciones impuestas por la pre y post condición, y luego redacte varias versiones de su traducción, donde en cada una de ella haya eliminado alguna frase de traducción literal de los operadores que aparecen en la especificación formal.

$f_1 : \mathsf{Int} \longrightarrow [\mathsf{Int}] \longrightarrow \mathsf{Bool}$

a) *Pre:* $0 < n \leq \#xs$

Post: $f_1.n.xs = (\forall\, i, j : 0 \leq i \wedge 0 \leq j \wedge i + j = n - 1 : xs.i = xs.j)$

$f_2 : \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow [\mathsf{Bool}] \longrightarrow \mathsf{Bool}$

b) *Pre:* $0 \leq i \leq j < \#xs \wedge \#xs > 0$

Post: $f_2.i.j.xs = ((\mathsf{N}\, k :\ i \leq k \leq j :\ xs.k) > 0)$

$f_3 : [\mathsf{Bool}] \longrightarrow \mathsf{Bool}$

Pre: *true*

c) *Post:* $f_3.xs \equiv (\mathsf{max}\ as, bs : [\mathsf{Bool}]\ :\ xs = as +\!\!+ bs \wedge (\forall i : 0 \leq i < \#as : as.i \equiv true)\ :\ \#as)$
$=$
$(\mathsf{N}\, k :\ 0 \leq k < \#xs :\ xs.k \equiv true)$

$f_4 : [\mathsf{Int}] \longrightarrow \mathsf{Int} \longrightarrow \mathsf{Bool}$

d) *Pre:* *true*

Post: $f_4.xs.M = ((\mathsf{max}\, i : 0 \leq i < \#xs : xs.i) \leq M)$