Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Introducción a la especificación de problemas

Problemas y Especificaciones

Inicialmente los problemas resolveremos con una computadora serán planteados como funciones. Es decir:

- Dados ciertos datos de entrada, obtendremos un resultado
- Más adelante en la materia, extenderemos el tipo de problemas que podemos resolver...

Definición (Especificación) de un problema

problema *nombre*(*parámetros*) : tipo de dato del resultado {
requiere *etiqueta*: { condiciones sobre los parámetros de entrada }
asegura *etiqueta*: { condiciones sobre los parámetros de salida }
}

- *nombre*: nombre que le damos al problema
  - será resuelto por una función con ese mismo nombre
- *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
  - Nombre del parámetro
  - Tipo de datos del parámetro
- *tipo de dato del resultado*: tipo de dato del resultado del problema (inicialmente especificaremos funciones)
  - En los asegura, podremos referenciar el valor devuelto con el nombre de res
- *etiquetas*: son nombres opcionales que nos servirán para nombrar declarativamente a las condiciones de los requiere o aseguras.

Definición (Especificación) de un problema

- *Sobre los requiere*
  - Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
  - Puede haber más de un requiere (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
  - Evitar contradicciones (un requiere no debería contradecir a otro).
- *Sobre los asegura*
  - Describen todas las condiciones y posibles valores o casuísticas de los parámetros de salida y entrada/salida en función de los parámetros de entrada.
  - Puede haber más de un asegura (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
  - Evitar contradicciones (un asegura no debería contradecir a otro).

¿Cómo contradicciones?

problema *soyContradictorio*$(x : \mathbb{Z}) : \mathbb{Z}\{$
  requiere esMayor: $\{x > 0\}$
  requiere esMenor: $\{x < 0\}$
  asegura esElSiguiente: $\{res + 1 = x\}$
  asegura esElAnterior: $\{res - 1 = x\}$
$\}$

Ejemplos

problema $raizCuadrada(x : \mathbb{R}) : \mathbb{R}$ {
requiere: $\{x \geq 0\}$
asegura: $\{res * res = x \land res \geq 0\}$
}

problema $sumar(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$ {
requiere: $\{True\}$
asegura: $\{res = x + y\}$
}

problema $restar(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$ {
requiere: $\{True\}$
asegura: $\{res = x - y\}$
}

problema $cualquieramayor(x : \mathbb{Z}) : \mathbb{Z}$ {
requiere: $\{True\}$
asegura: $\{res > x\}$
}

¿Por qué nuestro lenguaje será semiformal?: Ejemplos

problema *raizCuadrada*$(x : \mathbb{R}) : \mathbb{R}$ {
requiere: $\{x$ debe ser mayor o igual que $0\}$
asegura: $\{res$ debe ser mayor o igual que $0\}$
asegura: $\{res$ elevado al cuadrado será $x\}$
}

problema *sumar*$(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$ {
requiere: $\{-\}$
asegura: $\{res$ es la suma de x e $y\}$
}

problema *restar*$(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$ {
requiere: {Siempre cumplen}
asegura: $\{res$ es la resta de x menos $y\}$
}

problema *cualquieramayor*$(x : \mathbb{Z}) : \mathbb{Z}$ {
requiere: {Vale para cualquier valor posible de x}
asegura: $\{res$ debe tener cualquier valor mayor a $x\}$
}

El contrato

- **Contrato:** *El programador escribe un programa P tal que si el usuario suministra datos que hacen verdadera la precondición, entonces P termina en una cantidad finita de pasos retornando un valor que hace verdadera la postcondición.*
- El programa P es correcto para la especificación dada por la precondición y la postcondición exactamente cuando se cumple el contrato.
- Si el usuario no cumple la precondición y P se cuelga o no cumple la poscondición...
  - ¿El usuario tiene derecho a quejarse?
  - ¿Se cumple el contrato?
- Si el usuario cumple la precondición y P se cuelga o no cumple la poscondición...
  - ¿El usuario tiene derecho a quejarse?
  - ¿Se cumple el contrato?

Interpretando una especificación

- problema *raizCuadrada*($x : \mathbb{R}$) : $\mathbb{R}$ {
  requiere: {x debe ser mayor o igual que 0}
  asegura: {*res* debe ser mayor o igual que 0}
  asegura: {*res* elevado al cuadrado será x}
  }
- ¿Qué significa esta especificación?
- Se especifica que si el programa `raizCuadrada` se comienza a ejecutar en un estado que cumple $x \geq 0$, entonces el programa **termina** y el estado final cumple $res * res = x$ y $res \geq 0$.

Otro ejemplo

Dados dos enteros **dividendo** y **divisor**, obtener el cociente entero entre ellos.

problema $cociente(dividendo : \mathbb{Z}, divisor : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{divisor > 0\}$
  asegura: $\{res * divisor \leq dividendo\}$
  asegura: $\{(res + 1) * divisor > dividendo\}$
}

Qué sucede si ejecutamos con ...

- dividendo $= 1$ y divisor $= 0$?
- dividendo $= -4$ y divisor $= -2$, y obtenemos *res* $= 2$?
- dividendo $= -4$ y divisor $= -2$, y obtenemos *res* $= 0$?
- dividendo $= 4$ y divisor $= -2$, y el programa no termina?

Problemas comunes de las especificaciones

- ¿Qué sucede si especifico de menos?
- ¿Qué sucede si especifico de más?

Sobre-especificación

- Consiste en dar una postcondición más restrictiva de la que se necesita, o bien dar una precondición más laxa.
- Limita los posibles algoritmos que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.
- Ejemplo:

  problema $distinto(x : \mathbb{Z}) : \mathbb{Z}$ {

  requiere: $\{True\}$

  asegura: $\{res = x + 1\}$

  }
- ... en lugar de:

  problema $distinto(x : \mathbb{Z}) : \mathbb{Z}$ {

  requiere: $\{True\}$

  asegura: $\{res \neq x\}$

  }

Sub-especificación

- Consiste en dar una precondición más restrictiva de lo realmente necesario, o bien una postcondición más débil de la que se necesita.
- Deja afuera datos de entrada o ignora condiciones necesarias para la salida (permite soluciones no deseadas).
- Ejemplo:

  problema $distinto(x : \mathbb{Z}) : \mathbb{Z}\{$
  requiere: $\{x > 0\}$
  asegura: $\{res \neq x\}$
  $\}$

  ... en vez de:

  problema $distinto(x : \mathbb{Z}) : \mathbb{Z}\{$
  requiere: $\{True\}$
  asegura: $\{res \neq x\}$
  $\}$

Tipos de datos

- Un **tipo de datos** es un conjunto de valores (el conjunto base del tipo) provisto de una serie de operaciones que involucran a esos valores.

- Para hablar de un elemento de un tipo T en nuestro lenguaje, escribimos un término o expresión
  - Variable de tipo T (ejemplos: x, y, z, etc)
  - Constante de tipo T (ejemplos: 1, -1, $\frac{1}{5}$, 'a', etc)
  - Función (operación) aplicada a otros términos (del tipo T o de otro tipo)

- Todos los tipos tienen un elemento distinguido: $\perp$ o Indef

Tipos de datos de nuestro lenguaje de especificación

- Básicos
  - Enteros ($\mathbb{Z}$)
  - Reales ($\mathbb{R}$)
  - Booleanos (Bool)
  - Caracteres (Char)
- Enumerados
- Uplas
- Secuencias

Tipo $\mathbb{Z}$ (números enteros)

- Su conjunto base son los números enteros.
- Constantes: $0 \quad ; \quad 1 \quad ; \quad -1 \quad ; \quad 2 \quad ; \quad -2 \quad ; \quad \ldots$
- Operaciones aritméticas:
  - $a + b$ (suma); $a - b$ (resta); abs(a) (valor absoluto)
  - $a * b$ (multiplicación); a div b (división entera);
  - a mod b (resto de dividir a a por b), a^b o pot(a,b) (potencia)
  - a / b (división, da un valor de $\mathbb{R}$)
- Fórmulas que comparan términos de tipo $\mathbb{Z}$:
  - $a < b$ (menor)
  - $a \leq b$ o a <= b (menor o igual)
  - $a > b$ (mayor)
  - $a \geq b$ o a >= b (mayor o igual)
  - $a = b$ (iguales)
  - $a \neq b$ (distintos)

Tipo $\mathbb{R}$ (números reales)

- Su conjunto base son los números reales.
- Constantes: $0 \quad ; \quad 1 \quad ; \quad -7 \quad ; \quad 81 \quad ; \quad 7{,}4552 \quad ; \quad \pi \ldots$
- Operaciones aritméticas:
  - Suma, resta y producto (pero no div y mod)
  - a/b (división)
  - $\log_b(a)$ (logaritmo)
  - Funciones trigonométricas
- Fórmulas que comparan términos de tipo $\mathbb{R}$:
  - $a < b$ (menor)
  - $a \leq b$ o a `<=` b (menor o igual)
  - $a > b$ (mayor)
  - $a \geq b$ o a `>=` b (mayor o igual)
  - $a = b$ (iguales)
  - $a \neq b$ (distintos)

Tipo Bool (valor de verdad)

- Su conjunto base es $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$.
- Conectivos lógicos: !, &&, ||, con la semántica bi-valuada estándar.
- Fórmulas que comparan términos de tipo Bool:
  - $a = b$
  - $a \neq b$ (se puese escribir $a \,!= b$)

Tipo Char (caracteres)

- Sus elementos son las letras, dígitos y símbolos.
- Constantes: $`a`, `b`, `c`, \ldots, `z`, \ldots, `A`, `B`, `C`, \ldots, `Z`, \ldots,$ $`0`, `1`, `2`, \ldots, `9`$ (en el orden dado por el estándar ASCII).
- Función ord, que numera los caracteres, con las siguientes propiedades:
  - $\text{ord}(`a`) + 1 = \text{ord}(`b`)$
  - $\text{ord}(`A`) + 1 = \text{ord}(`B`)$
  - $\text{ord}(`1`) + 1 = \text{ord}(`2`)$
- Función char, de modo tal que si c es cualquier char entonces $\text{char}(\text{ord}(c)) = c$.
- Las comparaciones entre caracteres son comparaciones entre sus órdenes, de modo tal que $a < b$ es equivalente a $\text{ord}(a) < \text{ord}(b)$.

Tipos enumerados

- Cantidad finita de elementos.
  Cada uno, denotado por una constante.

  enum *Nombre* { *constantes* }

- *Nombre* (del tipo): tiene que ser nuevo.
- *Constantes*: nombres nuevos separados por comas.
- Convención: todos en mayúsculas.
- `ord`(a) da la posición del elemento en la definición (empezando de 0).
- Inversa: se usa el nombre del tipo funciona como inversa de `ord`.

Ejemplo de tipo enumerado

Definimos el tipo `Día` así:

```
enum Día {
  LUN, MAR, MIER, JUE, VIE, SAB, DOM
}
```

Valen:

- ord(LUN) $= 0$
- Día(2) $=$ MIE
- JUE $<$ VIE

Tipo upla (o tupla)

- Uplas, de dos o más elementos, cada uno de cualquier tipo.
- $T_0 \times T_1 \times \cdots \times T_k$: Tipo de las k-uplas de elementos de tipos T_0, T_1, ... T_k, respectivamente, donde k es fijo.
- Ejemplos:
  - $\mathbb{Z} \times \mathbb{Z}$ son los pares ordenados de enteros.
  - $\mathbb{Z} \times$ Char $\times$ Bool son las triplas ordenadas con un entero, luego un carácter y luego un valor booleano.
- *n*ésimo: $(a_0, \ldots, a_k)_m$ es el valor a_m en caso de que $0 \leq m \leq k$. Si no, está indefinido.
- Ejemplos:
  - $(7, 5)_0 = 7$
  - $(\text{‘a’}, DOM, 78)_2 = 78$

Secuencias

- **Secuencia:** Varios elementos del mismo tipo T, posiblemente repetidos, ubicados en un cierto orden.
- $seq\langle T\rangle$ es el tipo de las secuencias cuyos elementos son de tipo T.
- T es un tipo arbitrario.
  - Hay secuencias de $\mathbb{Z}$, de Bool, de `Días`, de 5-uplas;
  - también hay secuencias de secuencias de T;
  - etcétera.

Secuencias. Notación

- Una forma de escribir un elemento de tipo $\mathit{seq}\langle T\rangle$ es escribir términos de tipo T separados por comas, entre $\langle \dots \rangle$.
  - $\langle 1,2,3,4,1,0\rangle$ es una secuencia de $\mathbb{Z}$.
  - $\langle 1,\ 1+1,\ 3,\ 2*2,\ 5 \bmod 2,\ 0\rangle$ es otra secuencia de $\mathbb{Z}$ (igual a la anterior).
- La secuencia vacía se escribe $\langle\rangle$, cualquiera sea el tipo de los elementos de la secuencia.
- Se puede formar secuencias de elementos de cualquier tipo.
  - Como $\mathit{seq}\langle\mathbb{Z}\rangle$ es un tipo, podemos armar secuencias de $\mathit{seq}\langle\mathbb{Z}\rangle$ (secuencias de secuencias de $\mathbb{Z}$, o sea $\mathit{seq}\langle\mathit{seq}\langle\mathbb{Z}\rangle\rangle$).
  - $\langle\langle 12,13\rangle,\langle -3,9,0\rangle,\langle 5\rangle,\langle\rangle,\langle\rangle,\langle 3\rangle\rangle$ es un elemento de tipo $\mathit{seq}\langle\mathit{seq}\langle\mathbb{Z}\rangle\rangle$.

Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ($seq\langle\mathbb{Z}\rangle$, etc...)

- $\langle 1, 2, 3, 4, 5 \rangle$? Bien Formada. Tipa como $seq\langle\mathbb{Z}\rangle$ y $seq\langle\mathbb{R}\rangle$
- $\langle 1, 2, 3, 4, \frac{1}{0} \rangle$? No está bien formada porque uno de sus componentes está indefinido
- $\langle 1, true, 3, 4, 5 \rangle$? No está bien formada porque no es homogénea (Bool y $\mathbb{Z}$)
- $\langle 'a', 2, 3, 4, 5 \rangle$? No está bien formada porque no es homogénea ($Char$ y $\mathbb{Z}$)
- $\langle 'H', 'o', 'l', 'a' \rangle$? Bien Formada. Tipa como $seq\langle Char \rangle$
- $\langle true, false, true, true \rangle$? Bien Formada. Tipa como $seq\langle$Bool$\rangle$
- $\langle \frac{2}{5}, \pi, e \rangle$? Bien Formada. Tipa como $seq\langle\mathbb{R}\rangle$
- $\langle\rangle$? Bien formada. Tipa como cualquier secuencia $seq\langle X \rangle$ donde X es un tipo válido.
- $\langle\langle\rangle\rangle$? Bien formada. Tipa como cualquier secuencia $seq\langle seq\langle X \rangle\rangle$ donde X es un tipo válido.

Funciones sobre secuencias

Longitud

- Longitud: $length(a : seq\langle T\rangle) : \mathbb{Z}$
  - Representa la longitud de la secuencia a.
  - Notación: $length(a)$ se puede escribir como $|a|$ o como $a.length$.
- Ejemplos:
  - $|\langle\rangle| = 0$
  - $|\langle 'H', 'o', 'l', 'a'\rangle| = 4$
  - $|\langle 1, 1, 2\rangle| = 3$

Funciones con secuencias

I-ésimo elemento

- Indexación: $seq\langle T\rangle[i:\mathbb{Z}]:T$
  - Requiere $0 \leq i < |a|$.
  - Es el elemento en la i-ésima posición de a.
  - La primera posición es la 0.
  - Notación: $a[i]$.
  - Si no vale $0 \leq i < |a|$ se indefine.
- Ejemplos:
  - $\langle 'H',' o',' l',' a'\rangle[0] = 'H'$
  - $\langle 'H',' o',' l',' a'\rangle[1] = 'o'$
  - $\langle 'H',' o',' l',' a'\rangle[2] = 'l'$
  - $\langle 'H',' o',' l',' a'\rangle[3] = 'a'$
  - $\langle 1,1,1,1\rangle[0] = 1$
  - $\langle\rangle[0] = \perp$ (Indefinido)
  - $\langle 1,1,1,1\rangle[7] = \perp$ (Indefinido)

Funciones con secuencias

Pertenece

- Pertenece: $pertenece(x : T, s : seq\langle T\rangle) : Bool$
  - Es **true** sí y solo sí x es elemento de s.
  - Notación: $pertenece(x, s)$ se puede escribir como $x \in s$.
- Ejemplos:
  - $(1, MAR) \in \langle(1, LUN), (2, MAR), (3, JUE), (1, MAR)\rangle$? **true**
  - $(1, MAR) \in \langle(1, LUN), (2, MAR), (3, JUE), (3, MAR)\rangle$? **false**

Funciones con secuencias

Igualdad

Dos secuencias s_0 y s_1 (notación $s_0 = s_1$) son iguales si y sólo si

- Tienen la misma cantidad de elementos
- Dada una posición, el elemento contenido en la secuencia s_0 es igual al elemento contenido en la secuencia s_1.

Ejemplos:

- $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$? Sí
- $\langle \rangle = \langle \rangle$? Sí
- $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$? Sí
- $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle$? No
- $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 4, 5, 6 \rangle$? No
- $\langle 1, 2, 3, 5, 4 \rangle = \langle 1, 2, 3, 4, 5 \rangle$? No

Funciones con secuencias

Cabeza o Head

- Cabeza: $head(a : seq\langle T\rangle) : T$
  - Requiere $|a| > 0$.
  - Es el primer elemento de la secuencia a.
  - Es equivalente a la expresión $a[0]$.
  - Si no vale $|a| > 0$ se indefine.
- Ejemplos:
  - $head(\langle 'H', 'o', 'l', 'a'\rangle) = 'H'$
  - $head(\langle 1, 1, 1, 1\rangle) = 1$
  - $head(\langle\rangle) = \perp$ (Indefinido)

Funciones con secuencias

Cola o Tail

- Cola: $tail(a : seq\langle T\rangle) : seq\langle T\rangle$
  - Requiere $|a| > 0$.
  - Es la secuencia resultante de eliminar su primer elemento.
  - Si no vale $|a| > 0$ se indefine.
- Ejemplos:
  - $tail(\langle 'H', 'o', 'l', 'a' \rangle) = \langle 'o', 'l', 'a' \rangle$
  - $tail(\langle 1, 1, 1, 1 \rangle) = \langle 1, 1, 1 \rangle$
  - $tail(\langle \rangle) = \perp$ (Indefinido)
  - $tail(\langle 6 \rangle) = \langle \rangle$

Funciones con secuencias

Agregar al principio o addFirst

- Agregar cabeza: $addFirst(t : T, a : seq\langle T\rangle) : seq\langle T\rangle$
  - Es una secuencia con los elementos de a, agregándole t como primer elemento.
  - Es una función que no se indefine
- Ejemplos:
  - $addFirst('x', \langle 'H', 'o', 'l', 'a'\rangle) = \langle 'x', 'H', 'o', 'l', 'a'\rangle$
  - $addFirst(5, \langle 1, 1, 1, 1\rangle) = \langle 5, 1, 1, 1, 1\rangle$
  - $addFirst(1, \langle\rangle) = \langle 1\rangle$

Funciones con secuencias

Concatenación o concat

- Concatenación: $concat(a : seq\langle T\rangle, b : seq\langle T\rangle) : seq\langle T\rangle$
  - Es una secuencia con los elementos de a, seguidos de los de b.
  - Notación: $concat(a, b)$ se puede escribir $a +\!+\; b$.
- Ejemplos:
  - $concat(\langle 'H', 'o'\rangle, \langle 'l', 'a'\rangle) = \langle 'H', 'o', 'l', 'a'\rangle$
  - $concat(\langle 1, 2\rangle, \langle 3, 4\rangle) = \langle 1, 2, 3, 4\rangle$
  - $concat(\langle\rangle, \langle\rangle) = \langle\rangle$
  - $concat(\langle 2, 3\rangle, \langle\rangle) = \langle 2, 3\rangle$
  - $concat(\langle\rangle, \langle 5, 7\rangle) = \langle 5, 7\rangle$

Funciones con secuencias

Subsecuencia o subseq

- Subsecuencia: $subseq(a : seq\langle T\rangle, d, h : \mathbb{Z}) : seq\langle T\rangle$
  - Es una sublista de a en las posiciones entre d (inclusive) y h (exclusive).
  - Cuando $0 \leq d = h \leq |a|$, retorna la secuencia vacía.
  - Cuando no se cumple $0 \leq d \leq h \leq |a|$, se indefine!
- Ejemplos:
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, 0, 1) = \langle 'H'\rangle$
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, 0, 4) = \langle 'H', 'o', 'l', 'a'\rangle$
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, 2, 2) = \langle\rangle$
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, -1, 3) = \perp$
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, 0, 10) = \perp$
  - $subseq(\langle 'H', 'o', 'l', 'a'\rangle, 3, 1) = \perp$

Funciones con secuencias

- Cambiar una posición: $setAt(a : seq\langle T\rangle, i : \mathbb{Z}, val : T) : seq\langle T\rangle$
  - Requiere $0 \leq i < |a|$
  - Es una secuencia igual a a, pero con valor val en la posición i.
- Ejemplos:
  - $setAt(\langle 'H', 'o', 'l', 'a'\rangle, 0, 'X') = \langle 'X', 'o', 'l', 'a'\rangle$
  - $setAt(\langle 'H', 'o', 'l', 'a'\rangle, 3, 'A') = \langle 'H', 'o', 'l', 'A'\rangle$
  - $setAt(\langle\rangle, 0, 5) = \perp$ (Indefinido)

Operaciones sobre secuencias

- $length(a : seq\langle T\rangle) : \mathbb{Z}$ (notación $|a|$)
- $pertenece(x : T, s : seq\langle T\rangle)$: Bool (notación $x \in s$)
- indexación: $seq\langle T\rangle[i : \mathbb{Z}] : T$
- igualdad: $seq\langle T\rangle = seq\langle T\rangle$
- $head(a : seq\langle T\rangle) : T$
- $tail(a : seq\langle T\rangle) : seq\langle T\rangle$
- $addFirst(t : T, a : seq\langle T\rangle) : seq\langle T\rangle$
- $concat(a : seq\langle T\rangle, b : seq\langle T\rangle) : seq\langle T\rangle$ (notación a$++$b)
- $subseq(a : seq\langle T\rangle, d, h : \mathbb{Z}) : \langle T\rangle$
- $setAt(a : seq\langle T\rangle, i : \mathbb{Z}, val : T) : seq\langle T\rangle$

Predicados

- Asignan un nombre a una expresión.
- Facilitan la lectura y la escritura de especificaciones.
- Modularizan la especificación.

  pred p(argumentos)$\{f\}$

  - p es el nombre del puede usarse en el resto de la especificación en lugar de la formula f.

Ejemplos de Predicados

- pred $esPar(n : \mathbb{Z})\{$ (n mod 2) $= 0$ $\}$

  pred $esImpar(n : \mathbb{Z})\{$ $\neg$ (esPar(n)) $\}$

- pred $esFinde(d :$ Día$)\{d =$ SAB $\vee$ $d =$ DOM$\}$

  Otra forma:

  pred $esFinde2(d :$ Día$)\{d >$ VIE$\}$

- pred $tieneUnCinco(s : seq\langle\mathbb{Z}\rangle)\{$Alguno de los elementos de s es un 5$\}$

  Otra forma:

  pred $tieneUnCinco(s : seq\langle\mathbb{Z}\rangle)\{(\exists e : \mathbb{Z})\ e = 5 \wedge e \in s\}$

  Otra forma:

  pred $tieneUnCinco(s : seq\langle\mathbb{Z}\rangle)\{(\exists i : \mathbb{Z})\ 0 \leq i < |s| \wedge s[i] = 5\}$

Ejemplos de Predicados

- pred $todosImpares(s : seq\langle\mathbb{Z}\rangle)$ {Todos los elementos de s son impares}

  Otra forma:

  pred $todosImpares(s : seq\langle\mathbb{Z}\rangle)\{(\forall i : \mathbb{Z})\ i \notin s \lor esImpar(i)\}$

  Otra forma:

  pred $todosImpares(s : seq\langle\mathbb{Z}\rangle)\{(\forall i : \mathbb{Z})\ 0 \leq i < |s| \rightarrow esImpar(s[i])\}$

Expresiones condicionales

Función que elige entre dos elementos del mismo tipo, según una fórmula lógica (guarda)

- si la guarda es verdadera, elige el primero
- si no, elige el segundo

Por ejemplo

- expresión que devuelve el máximo entre dos elementos:

  problema $maximoEntreDos(a : \mathbb{Z}, b : \mathbb{Z}) : \mathbb{Z}$ {
  asegura: $\{res = \text{IfThenElseFi}\langle\mathbb{Z}\rangle(a > b, a, b)\}$
  }

  cuando los argumentos se deducen del contexto, se puede escribir directamente

  problema $maximoEntreDos(a : \mathbb{Z}, b : \mathbb{Z}) : \mathbb{Z}$ {
  asegura: $\{res = \text{if } a > b \text{ then } a \text{ else } b \text{ fi}\}$
  }
  problema $maximoEntreDos(a : \mathbb{Z}, b : \mathbb{Z}) : \mathbb{Z}$ {
  asegura: $\{res =$ es el mayor entre a y b $\}$
  }

Expresiones condicionales

Función que elige entre dos elementos del mismo tipo, según una fórmula lógica (guarda)

- si la guarda es verdadera, elige el primero
- si no, elige el segundo

Por ejemplo

- expresión que dado x un número entero, devuelve $1/x$ si $x \neq 0$ y 0 sino

  problema *unoSobre*$(x : \mathbb{Z}) : \mathbb{R}$ {
  asegura: $\{res = \text{if } x \neq 0 \text{ then } 1/x \text{ else } 0 \text{ fi}\}$
  }

Ejemplo (semiformal)

- **Ejemplo:** Crear un predicado `esPrimo` que sea **Verdadero** si y sólo si el número n es un número primo.

- pred esPrimo($n : \mathbb{Z}$) {
  n es mayor que 1 y sólo divisible por sí mismo y la unidad
  }

- **Ejemplo:** Especificar el problema de, dado un número mayor a 1, indicar si el número es un número primo.

- problema *primo*($n : \mathbb{Z}$) : Bool {
  requiere: $\{n > 1\}$
  asegura: $\{res = true \leftrightarrow \text{esPrimo}(n)\}$
  }

Ejemplo

- **Ejemplo:** Crear un predicado `esPrimo` que sea **Verdadero** si y sólo si el número n es un número primo.

- pred esPrimo($n : \mathbb{Z}$) {
  $n > 1 \ \wedge \ (\forall n' : \mathbb{Z})(1 < n' < n \rightarrow n \bmod n' \neq 0)$
  }

- **Observación:** `x mod y` se indefine si $y = 0$.

- **Ejemplo:** Especificar el problema de, dado un número mayor a 1, indicar si el número es un número primo.

- problema *primo*($n : \mathbb{Z}$) : Bool {
  requiere: $\{n > 1\}$
  asegura: $\{res = true \leftrightarrow \text{esPrimo}(n)\}$
  }

Modularizacion

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una una permutación[1] de la otra secuencia:
¿Cuándo será una secuencia permutación de la otra?

- Tienen los mismos elementos
- Cada elemento aparece la misma cantidad de veces en ambas secuencias

problema $esPermutacion(s1, s2 : seq\langle T\rangle) : \text{Bool}$ {
  asegura: $\{res = true \leftrightarrow ((\forall e : T)(cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))\}$
}

Pero... falta algo...

[1] mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

Modularizacion

Partiendo un problema en problemas mas chicos

Ahora, tenemos que especificar el problema *cantidadDeApariciones*

¿Cómo podemos saber la cantidad de apariciones de un elemento en una lista?

- Podríamos sumar 1 por cada posición donde el elemento en dicha posición es el que buscamos!
- Las operaciones de Sumatorias y Productorias también podemos usarlos

problema $cantidadDeApariciones(s : seq\langle T\rangle, e : T) : \mathbb{Z}$ {
  asegura $\{res = \sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})\}$
}

Recapitulando

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una una permutación[1] de la otra secuencia:

problema $esPermutacion(s1, s2 : seq\langle T\rangle) : \text{Bool}$ {
  asegura: $\{res = true \leftrightarrow ((\forall e : T)(cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))\}$
}

Donde...

problema $cantidadDeApariciones(s : seq\langle T\rangle, e : T) : \mathbb{Z}$ {
  asegura: $\{res = \sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})\}$
}

Y así podemos modularizar y descomponer nuestro problemas, partiendolos en problemas más chicos. Y también los podremos reutilizar!

[1] mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

Modularización

O partir el problema en problemas más chicos...

Los conceptos de modularización y encapsulamiento siempre estarán relacionados con los principios de diseño de software. La estrategia se puede resumir en:

- Descomponer un problema grande en problemas más pequeños (y sencillos)
- Componerlos y obtener la solución al problema original

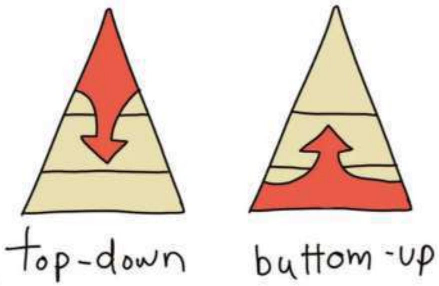
Esto favocere muchos aspectos de calidad como:

- La reutilización (una función auxiliar puede ser utilizada en muchos contextos)
- Es más facil probar algo chico que algo grande (si cada parte cumple su función correctamente, es más probable que todas juntas también lo haga)
- La declaratividad (es más facil entender al ojo humano)

Modularización

Top Down versus Bottom Up

También es aplicable a la especificación de problemas:



problema $esPermutacion(s1, s2 : seq\langle T\rangle) : \text{Bool}$ {
  asegura: $\{res = true \leftrightarrow ((\forall e : T)(cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))\}$
}

problema $cantidadDeApariciones(s : seq\langle T\rangle, e : T) : \mathbb{Z}$ {
  asegura: $\{res = \sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})\}$
}

¿Lo encaramos Top Down o Bottom Up?