Algoritmos y Estructuras de Datos

Especificación y Contratos

Departamento de Computación - FCEyN - UBA

2025

Repaso: Especificación, algoritmo, programa

1. **Especificación:** descripción del problema a resolver.
   - ¿Qué problema tenemos?
   - Habitualmente, dada en lenguaje formal.
   - Es un contrato que describe las propiedades de los datos de entrada y las propiedades de la solución.
2. **Algoritmo:** descripción de la solución escrita para humanos.
   - ¿Cómo resolvemos el problema?
3. **Programa:** descripción de la solución para ser ejecutada en una computadora.
   - También, ¿cómo resolvemos el problema?
   - Pero descripto en un lenguaje de programación.

Repaso: Especificación de problemas

- Una especificación es un contrato que define qué se debe resolver y qué propiedades debe tener la solución.
  - Define el qué y no el cómo.
- La especificación de un problema incluye un conjunto de parámetros: datos de entrada cuyos valores serán conocidos recién al ejecutar el programa.
- Además de cumplir un rol "contractual", la especificación del problema es insumo para las actividades de:
  1. testing: ya vieron algo en IP,
  2. verificación formal de corrección: lo vamos a ver en AED,
  3. derivación formal (construir un programa a partir de la especificación): no lo vemos acá.

Repaso: Contratos

- Una especificación es un contrato entre el programador de una función y el usuario de esa función.
- **Ejemplo**: calcular la raíz cuadrada de un número real.
  - ¿Cómo es la especificación (informalmente, por ahora) de este problema?
  - ¿Qué vamos a recibir? → un número no negativo.
    - Obligación del usuario: no puede proveer números negativos.
    - Derecho del programador: puede suponer que el argumento recibido no es negativo.
  - ¿Qué vamos a devolver? → la raiz cuadrada del número recibido.
    - Obligación del programador: debe calcular la raíz, siempre y cuando haya recibido un número no negativo
    - Derecho del usuario: puede suponer que el resultado va a ser correcto

Repaso: Partes de una especificación (contrato)

1. Encabezado
2. Precondición o cláusula "requiere"
   - Condición sobre los argumentos, que el programador da por cierta.
   - Especifica lo que requiere la función para hacer su tarea.
   - Por ejemplo: "el valor de entrada es un real no negativo"
3. Postcondición o cláusula "asegura"
   - Condición sobre el resultado, que debe ser cumplida por el programador siempre y cuando el usuario haya cumplido la precondición.
   - Especifica lo que la función asegura que se va a cumplir después de llamarla (si se cumplía la precondición).
   - Por ejemplo: "la salida es la raíz cuadrada del valor de entrada"

Definición (Especificación) de un problema

proc *nombre*(*parámetros*) : *salida* {
  requiere { *P* }
  asegura { *Q* }
}

- *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:
  - Tipo de pasaje (entrada, salida o entrada/salida)
  - Nombre del parámetro
  - Tipo de datos del parámetro
- *salida*: Opcional. Tipo de dato de la salida.
- *P* y *Q* son **predicados**, denominados la precondición y la postcondición del procedimiento.

Ejemplos

proc *raizCuadrada*(in $x : \mathbb{R}$) : $\mathbb{R}$ {
  requiere $\{x \geq 0\}$
  asegura $\{result * result = x \wedge result \geq 0\}$
}

proc *sumar*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere $\{True\}$
  asegura $\{result = x + y\}$
}

proc *restar*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere $\{True\}$
  asegura $\{result = x - y\}$
}

proc *cualquieramayor*(in $x : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere $\{True\}$
  asegura $\{result > x\}$
}

Ejemplo

Especificar un sensor de una pava eléctrica que dada una secuencia de temperaturas indica si el agua está lista para Café (todas las temperaturas entre $86 - 100°$, devuelve "C"), para Mate (todas las temperaturas entre $70 - 85°$, devuelve "M"), o para Ninguna (todas las temperaturas entre $0 - 69°$, devuelve "N")

Empecemos en lenguaje natural

proc *pavaElectrica*(*in s* : *seq* $< \mathbb{Z} >$) : *Char*{
  requiere {Todas las temperaturas entre 0 y 100 }
  asegura {res vale "C" cuando todos los elementos de *s* están entre 86 y 100}
  asegura {res vale "M" cuando todos los elementos de *s* están entre 70 y 85}
  asegura {res vale "N" cuando todos los elementos de *s* están entre 0 y 69}
}

Y ahora en lógica

proc *pavaElectrica*(*in s* : *seq* $< \mathbb{Z} >$) : *Char*{
  requiere { $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 100)$ }
  asegura { $res = `C` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 86 \leq s[i] \leq 100)$ }
  asegura { $res = `M` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 70 \leq s[i] \leq 85)$ }
  asegura { $res = `N` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 69)$ }
}

Interpretando una especificación

proc *pavaElectrica*(*in s* : *seq* $< \mathbb{Z} >$) : *Char*{
  requiere { $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 100)$ }
  asegura { $res = `C` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 86 \leq s[i] \leq 100)$ }
  asegura { $res = `M` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 70 \leq s[i] \leq 85)$ }
  asegura { $res = `N` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 69)$ }
}

Qué sucede si ejecutamos con ...

- s $=< 10, 2, 50 >$; res $= `N`$
- s $=< 83, 72, 80 >$; res $= `M`$
- s $=< 103, 80 >$; res $=$? Podria devolver cualquier cosa (Char), o colgarse, no es nuestra responsabilidad
- s $=< 5, 83 >$; res $=$? Tiene que devolver algo de tipo Char, pero no puede ser C, M, o N. ¿Estamos sub-especificando?
- s $=<>$; res $=$? Tiene que devolver algo de tipo Char, pero no puede ser C, M, o N. ¿Estamos sub-especificando?
- s $=< 80, 83 >$; res $= `J`$ ¿Es incorrecto para nuestra especificación? ¿Estamos sub-especificando? (en un rato lo vemos)

Pasaje de parámetros: in, inout

- Parámetros de entrada (**in**): Si se invoca el procedimiento con el argumento **c** para un parámetro de este tipo, entonces se copia el valor **c** antes de iniciar la ejecución
- Parámetros de entrada-salida (**inout**): Es un parámetro que es a la vez de entrada (se copia el valor del argumento al inicio), como de salida (se copia el valor de la variable al argumento). El efecto final es que la ejecución del procedimiento modifica el valor del parámetro.
- Todos los parámetros con atributo **in** (incluso **inout**) están inicializados

Argumentos que se modifican (inout)

Problema: Incrementar en 1 el argumento de entrada.

- Alternativa sin modificar la entrada (usual).

  proc incremento(in $a : \mathbb{Z}$) : $\mathbb{Z}${
  requiere $\{True\}$
  asegura $\{result = a + 1\}$
  }

- Alternativa que modifica la entrada: usamos el mismo argumento para la entrada y para la salida.

  proc incremento-modificando(inout $a : \mathbb{Z}$){
  requiere $\{A_0 = a\}$
  asegura $\{a = A_0 + 1\}$
  }

- En el requiere introducimos una metavariable (A_0) que representa el valor inicial de la variable a. La usamos en la postcondición para relacionar el valor de salida de a con su valor inicial.

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

Breve resumen de los tipos que tenemos disponibles, con sus símbolos de funciones y símbolos de predicado.
Para más detalles, ver apunte de especificación.

Tipos	Constantes	Funciones	Comparaciones
Enteros ($\mathbb{Z}$)	$0; -1; 1;$ $-2; 2; ...$	$+; -; abs(); *;$ $div(); mod(); a^b$	$<; \leq; >; \geq;$ $=; \neq$
Reales ($\mathbb{R}$)	$0; -1; 1;$ $\pi; ...$	$+; -; ...; abs();$ $log(); sin(); cos(); ...$	$<; \leq; >; \geq;$ $=; \neq$
Caracteres (Char)	'a'; 'b'; ...; '1'; '2'; ...	ord(), char()	$<; \leq; >; \geq;$ $=; \neq$ (comparan los ord())
Uplas	$T_0 x ... x T_n$	indexación: U_i	$=; \neq$
Secuencias	$< x_0, x_1, ... >$	indexación: $S[i]$; $length(S) =\mid S \mid$; $head(); tail();$ $concat(S1, S2) = S1 + S2$ $subseq(S, inicio, fin)$	$=; \neq$
Conjuntos			
Enumerados			

Funciones y predicados auxiliares

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

aux f(argumentos) : $tipo = e$;

- f es el nombre de la función, que puede usarse en el resto de la especificación en lugar de la expresión e.
- Los argumentos son opcionales y se reemplazan en e cada vez que se usa f.
- $tipo$ es el tipo del resultado de la función (el tipo de e).

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

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

Funciones y predicados auxiliares

Volvamos al ejemplo de la pava (por ahora seguimos evitando la sub-especificación):

proc *pavaElectrica*(*in* $s : seq < \mathbb{Z} >$) : *Char*{
  requiere { $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 100)$ }
  asegura { $res = `C` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 86 \leq s[i] \leq 100)$ }
  asegura { $res = `M` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 70 \leq s[i] \leq 85)$ }
  asegura { $res = `N` \rightarrow (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L 0 \leq s[i] \leq 69)$ }
}

En esta especificación estamos repitiendo mucho algunas cosas. Probemos encapsular algo en un pred:

pred *todosEnRango*($s : seq < \mathbb{R} >, min : \mathbb{Z}, max : \mathbb{Z}$){
  { $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L min \leq s[i] \leq max)$}

Y ahora podemos escribir:

proc *pavaElectrica*(*in* $s : seq < \mathbb{Z} >$) : *Char*{
  requiere { *todosEnRango*$(s, 0, 100)$ }
  asegura { $res = `C` \rightarrow$ *todosEnRango*$(s, 86, 100)$ }
  asegura { $res = `M` \rightarrow$ *todosEnRango*$(s, 70, 85)$ }
  asegura { $res = `N` \rightarrow$ *todosEnRango*$(s, 0, 69)$ }
}

Ejemplos de auxiliares (funciones y predicados)

- aux *suc*$(x : \mathbb{Z}) : \mathbb{Z} = x + 1$;
- aux $e() : \mathbb{R} = 2{,}7182$;
- aux *inverso*$(x : \mathbb{R}) : \mathbb{R} = 1/x$;
- aux $\#$*Apariciones*$(s : seq < T >, e : T) : \mathbb{Z}$: devuelve cuántas veces aparece e en s
- pred *esPar*$(n : \mathbb{Z})$ { (n mod 2) $= 0$ }

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

Predicando sobre secuencias

- Crear un predicado que sea **Verdadero** si y sólo si una secuencia de enteros sólo posee enteros mayores a 5.
- Solución:

```
pred seq_gt_five(s: seq⟨ℤ⟩) {
  (∀i : ℤ)(0 ≤ i < |s| →_L s[i] > 5)
}
```

- Crear un predicado que sea **Verdadero** si y sólo si hay algún elemento en la secuencia s que sea par y mayor que 5.
- Solución:

```
pred seq_has_elem_even_gt_five(s: seq⟨ℤ⟩) {
  (∃i : ℤ)(
    (0 ≤ i < |s| ∧_L ((s[i] mod 2 = 0) ∧ (s[i] > 5))
}
```

- **Nota:** Prestar atención al uso de $\wedge_L$ con el cuantificador existencial y $\rightarrow_L$ con el cuantificador universal.

Definir auxiliares versus especificar problemas

Definimos funciones y predicados auxiliares

- Expresiones del lenguaje, que se usan dentro de las especificaciones como reemplazos sintácticos.
  - Las funciones son de cualquier tipo.
- Dado que es un reemplazo sintáctico, ¡no se permiten definiciones recursivas!

Especificamos problemas

- Condiciones (el contrato) que debería cumplir un algoritmo para ser solución del problema.
- En una especificación dando la precondición y la postcondición con predicados de primer orden.
- No podemos usar otros problemas en la especificación.
  Sí podemos usar predicados y funciones auxiliares ya definidos.

$\sum$ - Sumatoria

El lenguaje de especificación provee formas de acumular resultados para los tipos numéricos $\mathbb{Z}$ y $\mathbb{R}$.
El término

$$\sum_{i=from}^{to} Expr(i)$$

retorna la suma de todas las expresiones $Expr(i)$ entre $from$ y to.
Es decir,

$$Expr(from) + Expr(from + 1) + \cdots + Expr(to - 1) + Expr(to)$$

Algunas condiciones:

- $Expr(i)$ debe ser un tipo numérico ($\mathbb{R}$ o $\mathbb{Z}$).
- $from \leq to$ (retorna 0 si no se cumple).
- $from$ y to es un **rango** (finito) de valores enteros, caso contrario se indefine.
- Si existe i tal que $from \leq i \leq to$ y $Expr(i) = \bot$, entonces toda la expresión se indefine!

$\sum$ - Ejemplos

Retornar la sumatoria de una secuencia s de tipo $seq\langle T\rangle$.
Solución:

$$\sum_{i=0}^{|s|-1} s[i]$$

Ejemplos:

- Si $s = \langle 1, 1, 3, 3\rangle$ retornará

$$s[0] + s[1] + s[2] + s[3] = 1 + 1 + 3 + 3 = 8$$

- Si $s = \langle\rangle$, entonces $from = 0$ y $to = -1$, por lo tanto retornará 0

$\sum$ - Ejemplos

Retornar la sumatoria de la posición 1 (únicamente) de la secuencia s.
Solución:

$$\sum_{i=1}^{1} s[i]$$

Ejemplos:

- Si $s = \langle 7, 11, 3, 3, 2, 4\rangle$ retornará $s[1] = 11$.
- Si $s = \langle 7\rangle$ la sumatoria se indefine ya que $s[1] = \bot$.

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:

  aux máx$(a, b : \mathbb{Z}) : \mathbb{Z} =$ IfThenElseFi$\langle\mathbb{Z}\rangle(a > b, a, b)$;

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

  aux máx$(a, b : \mathbb{Z}) : \mathbb{Z} =$ IfThenElseFi$(a > b, a, b)$;

- expresión que dado x devuelve $1/x$ si $x \neq 0$ y 0 sino

  aux unoSobre$(x : \mathbb{R}) : \mathbb{R} = \underbrace{\text{IfThenElseFi}(x \neq 0, 1/x, 0)}_{\text{no se indefine cuando } x = 0}$

$\sum$ - Ejemplo con condicional

Retornar la cantidad de apariciones de un elemento e en la secuencia s.
Solución:

aux #apariciones(s: $seq\langle T\rangle$, e: T): $\mathbb{Z} =$
$\sum_{i=0}^{|s|-1}$(IfThenElseFi$(s[i] = e, 1, 0)$);

Ejemplos:

- #apariciones($\langle 5, 1, 1, 1, 3, 3\rangle$, 1)=3
- #apariciones($\langle 5, 1, 1, 1, 3, 3\rangle$, 2)=0
- #apariciones($\langle 5, 1, 1, 1, 3, 3\rangle$, 3)=2
- #apariciones($\langle 5, 1, 1, 1, 3, 3\rangle$, 5)=1
- #apariciones($\langle\rangle$, 5)=0

$\prod$ - Productoria

El término

$$\prod_{i=from}^{to} Expr(i)$$

retorna el producto de todas las expresiones $Expr(i)$ entre *from* y *to*. Es decir,

$$Expr(from) * Expr(from + 1) * \cdots * Expr(to - 1) * Expr(to)$$

- $Expr(i)$ debe ser un tipo numérico ($\mathbb{R}$ o $\mathbb{Z}$).
- *from* y *to* define un rango de valores enteros (finito) y $from \leq to$ (retorna 1 si no se cumple).
- Si $Expr(i) = \bot$, toda la productoria se indefine.

$\prod$ - Ejemplos

Retornar la productoria de los elementos mayores a 0 de la secuencia s.

Solución:

$$\prod_{i=0}^{|s|-1} \text{IfThenElseFi}(s[i] > 0, s[i], 1)$$

Ejemplos:

- Si $s = \langle 7, 1, -3, 3, 2, -4 \rangle$ retornará

  $$s[0] * s[1] * 1 * s[3] * s[4] * 1 = 7 * 1 * 1 * 3 * 2 * 1 = 42$$

- Si $s = \langle -7 \rangle$ retornará 1.

Sobre-especificación

- Consiste en dar una postcondición más restrictiva que lo que se necesita.
- Limita los posibles algoritmos que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.
- Ejemplo:
  proc *distinto*(in $x : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere {*True*}
  asegura {$result = x + 1$}
  }
- ... en lugar de:
  proc *distinto*(in $x : \mathbb{Z}$) : $\mathbb{Z}${
  requiere{*True*}
  asegura {$result \neq x$}
  }

Sub-especificación

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

proc *distinto*(in $x : \mathbb{Z}) : \mathbb{Z}\{$
  requiere $\{x > 0\}$
  asegura $\{result \neq x\}$
$\}$

... en vez de:

proc *distinto*(in $x : \mathbb{Z}) : \mathbb{Z}\{$
  requiere $\{True\}$
  asegura $\{result \neq x\}$
$\}$

Sub-especificación

Volvamos de nuevo al ejemplo de la pava eléctrica, que dijimos que estaba sub-especificado:

proc *pavaElectrica*(*in s* : *seq* $< \mathbb{Z} >) :$ *Char*$\{$
  requiere $\{$ *todosEnRango*$(s, 0, 100)$ $\}$
  asegura $\{$ *res* $=$ '*C*' $\rightarrow$ *todosEnRango*$(s, 86, 100)$ $\}$
  asegura $\{$ *res* $=$ '*M*' $\rightarrow$ *todosEnRango*$(s, 70, 85)$ $\}$
  asegura $\{$ *res* $=$ '*N*' $\rightarrow$ *todosEnRango*$(s, 0, 69)$ $\}$
$\}$

¿Cómo podemos especificarlo un poco mejor?

proc *pavaElectrica*(*in s* : *seq* $< \mathbb{Z} >) :$ *Char*$\{$
  requiere $\{$ *todosEnRango*$(s, 0, 100)$ $\}$
  asegura $\{$ *res* $=$ '*C*' $\leftrightarrow$ *todosEnRango*$(s, 86, 100)$ $\}$
  asegura $\{$ *res* $=$ '*M*' $\leftrightarrow$ *todosEnRango*$(s, 70, 85)$ $\}$
  asegura $\{$ *res* $=$ '*N*' $\leftrightarrow$ *todosEnRango*$(s, 0, 69)$ $\}$
$\}$

- Notar que sigue faltaldo qué hacer cuando hay mezclados

Especificaciones y comentarios

- Los nombres de los predicados/funciones ayudan a describir el significado de las precondiciones y postcondiciones de las especificaciones.
- Los comentarios (/*...*/) también ayudan a describir el significado de las precondiciones y postcondiciones de las especificaciones y son útiles si no hay predicados

Ejemplo:

proc menorElemDistintos(in s: *seq*$\langle\mathbb{Z}\rangle$): $\mathbb{Z}$ {
  requiere { noNegativos(s) $\wedge$ noHayRepetidos(s) }
  asegura {
    /* result es un indice válido de s */
    $0 \leq result < |s| \wedge_L$
    /* s[result] es el menor elemento de s */
    $(\forall i : \mathbb{Z})((0 \leq i < |s|) \rightarrow_L s[result] \leq s[i])$}
}

Ejemplo de la Meseta Maximal

Dada una secuencia de números enteros (alturas) se desea obtener como resultado una tupla que indique la altura y el largo de la meseta (números iguales consecutivos en la secuencia) máxima.

Algún problema con este enunciado? Un poco ambiguo? máxima?

proc MesetaMaximal(*in* $s : seq < \mathbb{Z} >$) : $\mathbb{Z}x\mathbb{N}$ {
  requiere { La secuencia tiene al menos dos elementos }
  asegura { existe una meseta en s, tal que $result_0$ es la
    altura y $result_1$ es el largo, y no hay otra meseta más larga }

Y ahora en lógica

proc MesetaMaximal(*in* $s : seq < \mathbb{Z} >$) : $\mathbb{Z}x\mathbb{N}$ {
  requiere { $| s | \geq 1$ }
  asegura { $(\exists s' : seq < \mathbb{Z} >)(s' \neq \langle\rangle \wedge_L esMeseta(s') \wedge esSubsecuencia(s', s)) \wedge$
    $result_0 = s'[0] \wedge result_1 =| s' | \wedge$
    $((\forall s'' : seq < \mathbb{Z} >)(esMeseta(s'') \wedge esSubsecuencia(s'', s) \rightarrow$
    $| s'' | \leq | s' |))$
  }

Ven problemas mirando la espec.?

Qué pasa con la secuencia vacía? O si no hay mesetas? O si hay más de una meseta de la misma longitud?

Ejemplo de la Meseta Maximal

proc MesetaMaximal(*in* $s : seq < \mathbb{Z} >$) : $\mathbb{Z}x\mathbb{N}$ {
  requiere { $| s | \geq 1$ }
  asegura { $(\exists s' : seq < \mathbb{Z} >)(s' \neq \langle\rangle \wedge_L esMeseta(s') \wedge esSubsecuencia(s', s)) \wedge$
    $result_0 = s'[0] \wedge result_1 =| s' | \wedge$
    $((\forall s'' : seq < \mathbb{Z} >)(esMeseta(s'') \wedge esSubsecuencia(s'', s) \rightarrow$
    $| s'' | \leq | s' |))$
  }

Además, faltan las auxiliares

pred *esMeseta*($s : seq < \mathbb{Z} >$){
  $(\forall i : \mathbb{N})(0 < i <| s | \rightarrow_L s[i] = s[0])$}

pred *esSubSecuencia*($r, s : seq < \mathbb{Z} >$){
  $(\exists s', s'' : seq\langle\mathbb{Z}\rangle)(s = s' + r + s'')$}

Bibliografía

- David Gries - The Science of Programming
  - Chapter 4 - Predicates (cuantificación, variables libres y ligadas, etc.)
  - Chapter 5 - Notations and Conventions for Arrays (secuencias)