Algoritmos y Estructuras de Datos

Apunte
Especificación de problemas

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

En esta parte de la materia nos dedicamos a *especificar* problemas. Para eso planteamos *procedimientos* que reciben datos de entrada y modifican algunos de ellos o devuelven datos de salida. Describiremos con un lenguaje formal las propiedades que tienen que cumplir los datos de entrada para que el programa se comporte adecuadamente (los *requiere*) y las propiedades que cumplirán los datos de salida (los *asegura*). Este documento contiene el detalle del lenguaje que vamos a usar para esta tarea.

1. Especificación de procedimientos

La definición de un procedimiento tiene tres partes:

- **Signatura.** Incluye el **nombre** del procedimiento, la lista de **parámetros** y el tipo de datos del **resultado**, si lo hubiera
- **Precondición.** Compuesta por las cláusulas *requiere*
- **Postcondición**. Compuesta por las cláusulas *asegura*

Veamos un ejemplo:

```
proc raizCuadrada(in x: ℝ): ℝ
     requiere {x > 0}
     asegura {res · res = x}
```

Esta especificación describe el comportamiento del procedimiento `raizCuadrada`, el cual recibe un dato de tipo real (`x`), que debe ser positivo ($x > 0$), y devuelve un valor de tipo real (*res*), que debe ser igual a la raíz cuadrada del valor de la entrada ($res \cdot res = x$).

1.1. Tipos de parámetros

Los parámetros de un procedimiento pueden ser de entrada (`in`) o entrada/salida (`inout`).

Si el procedimiento devuelve un valor se indica el tipo de dato del resultado por fuera de la lista de parámetros como resultado del procedimiento. Nos referiremos a este valor con la palabra reservada *res*. No tiene sentido referirse a su valor en los requiere.

```
proc doble(in a: ℤ): ℤ
     asegura {res = 2 · a}
```

Los parámetros de entrada (`in`) tienen un valor que puede leerse en cualquier momento y que no podrá ser modificado por el código del procedimiento, por lo que a la salida tendrá el mismo valor que a la entrada. Por lo tanto, es posible referirse al valor del mismo tanto en los requiere como en los asegura.

Los parámetros de entrada/salida (`inout`) pueden ser modificados por el procedimiento, pudiendo tener un valor de salida distinto al recibido en la entrada. Para referirnos al valor inicial utilizamos *metavariables*. Sea a una variable de tipo *inout*, definimos en el requiere el valor de una variable adicional (A_0) que preserva su valor y sobre la que podemos hacer referencia en los asegura para referirnos al valor inicial de a.

```
proc sumarUno(inout a: ℤ)
     requiere {a = A₀}
     asegura {a = A₀ + 1}
```

En este ejemplo, la variable a tiene a la entrada un valor desconocido que denominaremos A_0. A la salida el valor de a será igual al que tenía a la entrada más uno.

Es posible escribir muchas expresiones *requiere* y *asegura*. Las mismas se considerarán unidas con el conector $\wedge_L$ en el orden que fueron introducidas. A continuación se muestran dos especificaciones equivalentes.

```
proc recortarRango(inout s: seq<ℤ>, in desde: ℤ, in hasta: ℤ)
     requiere {0 ≤ desde < |s|}
     requiere {0 ≤ hasta < |s|}
     requiere {desde ≤ hasta}
     asegura {···}

proc recortarRango(inout s: seq<ℤ>, in desde: ℤ, in hasta: ℤ)
     requiere {0 ≤ desde < |s| ∧_L 0 ≤ hasta < |s| ∧_L desde ≤ hasta}
     asegura {···}
```

1.2. Predicados y funciones auxiliares

Para simplificar la escritura de predicados y facilitar su lectura y comprensión, es posible descomponerlos en funciones y predicados auxiliares.

Los predicados encapsulan fórmulas lógicas, y se evalúan a un valor de verdad. Por otro lado, las funciones auxiliares permiten encapsular una operatoria que tiene como resultado un elemento de un tipo específico. Veamos un ejemplo:

```
proc distanciaEntrePuntos2D(in x1: ℤ, in y1: ℤ, in x2: ℤ, in y2: ℤ): ℤ
     requiere {EsPositivo(x1) ∧ EsPositivo(y1) ∧ EsPositivo(x2) ∧ EsPositivo(y2)}
     asegura {res = Dist(x1, y1, x2, y2)}

pred EsPositivo(x: ℤ)
      {x > 0}

aux Dist(x1: ℤ, y1: ℤ, x2: ℤ, y2: ℤ): ℤ =
      sqrt((x2 − x1)^2 + (y2 − y1)^2)
```

Nótese que a diferencia de los procedimientos, los predicados y funciones auxiliares *no describen problemas*. Son simples herramientas sintácticas para descomponer predicados. Nunca modifican los parámetros, por lo que no es necesario indicar el tipo de pasaje (*in* o *inout*) como lo hacíamos con los *procs*.

El predicado anterior es *equivalente* a reemplazar el cuerpo en el predicado que lo referencia:

```
proc distanciaEntrePuntos(in x1: ℤ, in y1: ℤ, in x2: ℤ, in y2: ℤ): ℤ
     requiere {x1 ≥ 0 ∧ y1 ≥ 0 ∧ x2 ≥ 0 ∧ y2 ≥ 0}
     asegura {res = sqrt((x2 − x1)^2 + (y2 − y1)^2)}
```

Es muy importante notar que **no se puede utilizar una referencia a un procedimiento desde los requiere o asegura de otro procedimiento**. Tampoco desde un predicado auxiliar. El siguiente ejemplo es incorrecto.

```
proc máximo(in s: seq<ℤ>): ℤ
     requiere {···}
     asegura {···}

proc posiciónMaximo(in s: seq<ℤ>): ℤ
     requiere {|s| > 0}
     asegura {s[res] = máximo(s)} // INCORRECTO
```

1.3. Cuantificadores y funciones especiales

Para escribir los predicados de los requiere y asegura usaremos lógica trivaluada de primer orden, tal cuál se vio en la teórica. Los **cuantificadores** que usaremos son los siguientes:

Operación	Sintaxis	Significado
cuantificador universal	$(\forall i : T)(P(i))$	Todo valor i de tipo T tiene que cumplir el predicado $P(i)$
cuantificador existencial	$(\exists i : T)(P(i))$	Existe al menos un valor i de tipo T que cumple el predicado $P(i)$

Algunos ejemplos:

- $(\forall n : \mathbb{Z})(n \cdot n \geq n)$
  Todo número entero cumple que su cuadrado es mayor o igual a sí mismo.
- $(\forall n : \mathbb{Z})(n \bmod 4 = 0 \rightarrow n \bmod 2 = 0)$
  Todo número entero cumple que si es divisible por 4, entonces es divisible por 2.
- $(\exists i : \mathbb{Z})(10 \bmod i = 0)$
  Existe un número entero que cumple que 10 es divisible por él.

Es posible cuantificar sobre múltiples variables y anidar cuantificadores.

- $(\forall n, m : \mathbb{Z})((n > 0 \land m > 0 \land n < m) \rightarrow (n^2 < m^2))$
  Para todos dos números positivos n y m que cumplen con que n es menor a m, entonces el cuadrado de n es menor que el cuadrado de m.
- $(\forall n : \mathbb{Z})((\exists m : \mathbb{Z})(m < n))$
  Para todo número entero n, siempre existe un número m que es menor.
- $(\exists n : \mathbb{Z})((\forall m : \mathbb{Z})(m \bmod n = 0))$
  Existe un número entero n que divide a todos los números enteros.

Muy frecuentemente vamos a usar cuantificadores para describir el contenido de **secuencias**. Por ejemplo:

- $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L s[i] > 0)$
  Los elementos en todas las posiciiones de la secuencia s son mayores que cero. Usamos el operador $\rightarrow_L$ porque en caso de que se haga falso el antecedente (i no está en el rango) el consecuente podría indefinirse, ya que se intenta indexar en una posición inválida de la secuencia s.
- $(\forall n : \mathbb{Z})(n \in s \rightarrow n > 0)$
  Este predicado es equivalente al anterior pero en lugar de utilizar índices para recorrer la secuencia usamos *pertenece* ($\in$). Notar que ya no es necesario usar el $\rightarrow_L$ porque ahora no hay nada que se pueda indefinir en el consecuente.
- $(\exists i : \mathbb{Z})(0 \leq i < |s| \land i \bmod 2 = 0 \land_L s[i] \bmod 3 = 0)$
  Existe una posición par de la secuencia s que contiene un elemento divisible por 3.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow_L ((\exists k : \mathbb{Z})(k.k = s[i])))$
  Todos los elementos de la secuencia s son cuadrados perfectos.

Una **función especial** que usaremos es la función *IfThenElse*, o `if` *cond* `then` val_1 `else` val_2. Esta función evalúa una condición y devuelve el primer valor si la condición es verdadera y el segundo si la condición es falsa. La condición puede ser cualquier predicado y los dos valores deben ser del mismo tipo. Nótese que el tipo de la expresión completa será el mismo que el de los valores.

Algunos ejemplos:

- $IfThenElse(x > 0, x, -x)$
  Devuelve el valor absoluto de x.
- $IfThenElse(x1 > x2, x1 - x2, x2 - x1) + IfThenElse(y1 > y2, y1 - y2, y2 - y1)$
  Devuelve la distancia de Manhatan (sobre una grilla) entre los puntos $(x1, y1)$ y $(x2, y2)$.

Nótese que esta función *no se utiliza* para describir causalidad (si pasa P entonces se cumple Q), ya que la evaluación de la función *if* devuelve un valor de algún tipo. La forma correcta de expresar causalidad es utilizando la implicación, de la forma $P \to Q$.

Por último, para operar con los elementos de secuencias, vamos a usar los siguientes **operadores especiales**:

Operación	Sintaxis	Significado
sumatoria	$\sum_{i=j}^{n} f(i)$	Equivalente a $f(j) + f(j+1) + \cdots + f(n)$
productoria	$\prod_{i=j}^{n} f(i)$	Equivalente a $f(j).f(j+1).\cdots.f(n)$

Ejemplos:

- $\sum_{i=0}^{100} \ i$
  La suma de todos los enteros entre 0 y 100.
- $\prod_{i=1}^{10} \ 2 \cdot i$
  El producto de los primeros 10 números pares.
- $\sum_{i=0}^{|s|-1} \ s[i]$
  La suma de todos los elementos de la secuencia s.

Si se combinan estos operadores con el operador *if* se pueden agregar condiciones o incluso contar los elementos que cumplan una determinada condición:

- $\sum_{i=0}^{|s|-1} \ IfThenElse(s[i] \bmod 2 = 0, 1, 0)$
  La cantidad de elementos pares en la secuencia s.

Aclaración. Para cualquiera de los dos casos, sean a y b los límites inferior y superior respectivamente de los operadores, si ocurre que $a > b$ (coloquialmente denominado rango vacío) tenemos que se reducen al neutro de la operación que representan. Eso se da independientemente de la $f(i)$ que se encuentre dentro del operador.

$$\sum_{i=a}^{b} (\dots) = 0 \qquad\qquad \prod_{i=a}^{b} (\dots) = 1$$

2. Tipos de especificación

Resumimos aquí los tipos de datos que podremos usar para especificar. Asimismo, indicamos sus operaciones y su notación.

2.1. Tipos básicos

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

`bool`: valor booleano.

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

`int`: número entero.

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

`real` o `float`: número real.

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

`char`: caracter.

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

2.2. Tipos complejos

`seq<T>`: secuencia de tipo T.

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

- secuencias por extensión: permite crear secuencias a partir de sus elementos
  - sintaxis: $\langle x : T, y : T, z : T, \cdots\rangle : seq\langle T\rangle$
  - ejemplos: $\langle\rangle$ (secuencia vacía). $\langle 1, 2, 25\rangle$ (secuencia de enteros con tres elementos: 1, 2 y 25)
- longitud: devuelve la cantidad de elementos de la secuencia
  - sintaxis: $length(s : seq\langle T\rangle) : \mathbb{Z}$. Alternativas: $|s|$ o o $s.length$
  - ejemplos: $|\langle\rangle| = 0$, $|\langle 1, 2, 25\rangle| = 3$
- pertenece: indica si un elemento está en la secuencia
  - sintaxis: $i : T \in s : seq\langle T\rangle$ (devuelve un valor de verdad)
  - ejemplos: $1 \in \langle\rangle$ es falso, $1 \in \langle 2, 3\rangle$ es falso, $1 \in \langle 1, 2, 3\rangle$ es verdadero
- indexación: devuelve el elemento en una determinada posición de la secuencia. El primer elemento es el 0. Se indefine si el índice es menor a cero o mayor al tamaño de la secuencia.
  - sintaxis: $seq\langle T\rangle[i : \mathbb{Z}] : T$.
  - ejemplos: $\langle 1, 2, 3\rangle[0] = 1$, $\langle 1, 2, 3\rangle[2] = 3$, $\langle 1, 2, 3\rangle[5]$ es indefinido
- cabeza: devuelve el primer elemento de la secuencia. Se indefine si la secuencia es vacía
  - sintaxis: $head(s : seq\langle T\rangle) : T$
  - ejemplos: $head(\langle 1, 2, 3\rangle) = 1$, $head(\langle\rangle)$ es indefinido
- cola: devuelve la secuencia sin el primer elemento. Se indefine si la secuencia es vacía
  - sintaxis: $tail(s : seq\langle T\rangle) : seq\langle T\rangle$

- ejemplos: $tail(\langle 1,2,3\rangle) = \langle 2,3\rangle$, $tail(\langle 1\rangle) = \langle\rangle$, $tail(\langle\rangle)$ es indefinido

- concatenación: concatena dos secuencias
  - sintaxis: $concat(s_1 : seq\langle T\rangle, s_2 : seq\langle T\rangle) : seq\langle T\rangle$. Alternativa: $s_1 + +s_2$
  - ejemplos: $concat(\langle 1,2\rangle, \langle 3,4\rangle) = \langle 1,2,3,4\rangle$, $\langle\rangle$ ++ $\langle 1,2\rangle = \langle 1,2\rangle$
- subsecuencia: devuelve una subsecuencia de la secuencia original, incluyendo el índice del principio y sin incluir el índice del final. Se indefine si los índices están fuera de rango o si el índice del final es menor al del principio. Devuelve una secuencia vacía si los índices son iguales.
  - sintaxis: $subseq(s : seq\langle T\rangle, i : \mathbb{Z}, j : \mathbb{Z}) : seq\langle T\rangle$
  - ejemplos: $subseq(\langle 1,2,3,4,5\rangle, 1, 3) = \langle 2,3\rangle$, $subseq(\langle 1,2,3,4,5\rangle, 1, 1) = \langle\rangle$, $subseq(\langle 1,2,3,4,5\rangle, 3, 1)$ es indefinido
- cambiar elemento: devuelve una secuencia igual a la original pero con un elemento cambiado. Se indefine si el índice está fuera de rango.
  - sintaxis: $setAt(s : seq\langle T\rangle, i : \mathbb{Z}, val : T) : seq\langle T\rangle$
  - ejemplos: $setAt(\langle 1,2,3\rangle, 1, 5) = \langle 1,5,3\rangle$, $setAt(\langle 1,2,3\rangle, 5, 5)$ es indefinido

`string`: renombre de `seq<char>`.

`tupla<T1, ..., Tn>`: tupla de tipos $T_1, \ldots, T_n$

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

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

`struct<campo1: T1, ..., campon: Tn>`: tupla con nombres para los campos.

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

- struct por extensión: permite crear structs a partir de sus campos
  - sintaxis: $\langle campo_1 : T_1, campo_2 : T_2, \cdots\rangle : struct < campo_1 : T_1, campo_2 : T_2, \cdots >$
  - ejemplos: $\langle x : 20, y : 10\rangle$ (struct con dos campos de tipo $\mathbb{Z}$ denominados x e y con los valores 20 y 10 respectivamente
- obtener el valor de un campo: devuelve el valor de un campo de la struct. Se indefine si el campo no existe.
  - sintaxis: $struct < campo_1 : T_1, \ldots, campo_n : T_n > .campo_i : T_i$
  - ejemplos: $\langle x : 20, y : 10\rangle.x = 20$, $\langle x : 20, y : 10\rangle.z$ es indefinido