Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Especificación - clase 3

Lenguaje de especificación

Funciones auxiliares

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

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

- f es el nombre de la función, que puede usarse en el resto de la especificación en lugar de la expresión e
- Los parámetros 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)
- Ejemplo:

  aux $suc(x : \text{Int}) : \text{Int} = x + 1;$

Definir funciones auxiliares vs. especificar problemas

Definimos funciones auxiliares

- Es una expresión del lenguaje, que se usa dentro de las especificaciones como un reemplazo sintáctico.

Especificamos problemas

- Condiciones (el contrato) que debería cumplir alguna función para ser solución del problema.
- No quiere decir que exista esa función o que sepamos cómo hacer un algoritmo que la computa.
- En la especificación de un problema o de un tipo no podemos usar otro problema que hayamos especificado. Sí podemos usar las funciones auxiliares ya definidas.

Tipos enumerados

- Primer mecanismo para definir tipos propios dados por una cantidad finita de elementos. Cada uno está denotado por una constante.

  tipo *Nombre* = *constantes*;

- *Nombre* (del tipo): tiene que ser nuevo
- *constantes*: nombres nuevos separados por comas
- **Convención:** todos con mayúsculas
- ord(a) da la posición del elemento en la definición (empezando de 0)
- Inversa: nombre(i), o bien ord$^{-1}(i)$

Ejemplo de tipo enumerado

tipo Día = Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo;

- Valen:
  - ord(Lunes) == 0
  - Día(2) == Miércoles
  - Jueves < Viernes
- Podemos definir:

  aux *esFinde*(*d* : Día) : Bool =
  (*d* == Sábado || *d* == Domingo);
- Otra forma:

  aux *esFinde2*(*d* : Día) : Bool = *d* > Viernes;

Tipo upla

- Una *k*-upla es una secuencia de *k* elementos, cada uno puede ser de un tipo distinto
- Las uplas son tipos primitivos del lenguaje de especificación
- $(T_1, T_2, \ldots T_n)$: Tipo de las *n*-uplas de elementos. El *i*-ésimo elemento es de tipo T_i, para $i = 1, \ldots, n$. El valor de *n* es fijo para el tipo
- Por ejemplo, (Int, Int) son los pares ordenados de enteros, y *prm* y *sgd* devuelven sus componentes
- Ejemplo: $prm((7,5)) == 7$

Secuencias

- **Secuencia:** varios elementos del mismo tipo, posiblemente repetidos, ubicados en un cierto orden
- También se llaman listas, y son una familia de tipos
  - Para cada tipo de datos hay un tipo secuencia, que tiene como elementos las secuencias de elementos de ese tipo
- Las secuencias son muy importantes en el lenguaje de especificación
- $[T]$: Tipo de las secuencias cuyos elementos son de tipo T

Notación

- Una forma de escribir un elemento de tipo secuencia de tipo T es escribir varios términos de tipo T separados por comas, entre corchetes.
- Ejemplo: $[1, 1+1, 3, 2*2, 3, 5]$ es una secuencia de Int.
- La secuencia vacía (de elementos de cualquier tipo) se representa [ ].
- Se pueden formar secuencias de elementos de cualquier tipo
  - Como las secuencias de enteros son tipos, existen por ejemplo las secuencias de secuencias de enteros.
- Ejemplo: Una secuencia de secuencias de enteros ...

$$[[12, 13], [-3, 9, 0], [5], [], [], [3]]$$

Secuencias por comprensión

$$[\textit{expresión} \mid \textit{selectores},\ \textit{condiciones}]$$

- *expresión*: cualquier expresión válida del lenguaje
- *selectores*: *variable* $\leftarrow$ *secuencia* (se puede usar $\in$)
  - La *variable* va tomando el valor de cada elemento de la *secuencia*
  - Las variables que aparecen en selectores se llaman ligadas, el resto de las que aparecen en una expresión se llaman libres
- *condiciones*: expresiones de tipo Bool
- El resultado es una secuencia con el valor de la expresión calculado para todos los elementos seleccionados por los selectores que cumplen las condiciones.
- Ejemplo: $[(x,y)|x \leftarrow [1,2], y \leftarrow [1,2,3], x < y] == [(1,2),(1,3),(2,3)]$

Secuencias por comprensión, varios selectores

- Las siguientes expresiones son distintas:

$$[(x,y)|x \leftarrow a, y \leftarrow b, x \neq y]$$
$$[(x,y)|y \leftarrow b, x \leftarrow a, x \neq y]$$

- El orden importa!
- Tenemos ... $[(x,y)|x \leftarrow [1,2], y \leftarrow [3,4]] == [(1,3),(1,4),(2,3),(2,4)]$
- Pero ... $[(x,y)|y \leftarrow [3,4], x \leftarrow [1,2]] == [(1,3),(2,3),(1,4),(2,4)]$

Intervalos

$$[\textit{expresión}_1..\textit{expresión}_2]$$

- Las expresiones tienen que ser del mismo tipo, discreto y totalmente ordenado (por ejemplo, Int, Char, enumerados)
- **Resultado:** todos los valores del tipo entre el valor de $\textit{expresión}_1$ y el valor de $\textit{expresión}_2$ (ambos inclusive)
- Si no vale $\textit{expresión}_1 \leq \textit{expresión}_2$, la secuencia es vacía
- Con un paréntesis en lugar de un corchete, se excluye el extremo correspondiente

Ejemplos:

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

Ejemplos de secuencias por comprensión

Ejemplo 1: Divisores comunes a dos enteros positivos a y b dados

$$\text{aux } \textit{divCom}(a,b : \text{Int}) : [\text{Int}] = [x|x \leftarrow [1..a+b], \textit{divide}(x,a), \textit{divide}(x,b)];$$

$$\text{aux } \textit{divide}(a,b : \text{Int}) : \text{Bool} = b \bmod a == 0;$$

Ejemplo:

$$\textit{divCom}(8,12) == [1,2,4]$$

¿Cuáles son variables libres y cuáles ligadas?

Ejemplos de secuencias por comprensión

Ejemplo 2: Cuadrados de los elementos impares de una secuencia de enteros

$$\text{aux } cuadImp(a : [\text{Int}]) : [\text{Int}] = [x * x | x \leftarrow a, \neg divide(2, x)];$$

Ejemplo:

$$cuadImp([1..9)) == [1, 9, 25, 49]$$

Ejemplos de secuencias por comprensión

Ejemplo 3: Suma de los elementos pares distintos de dos secuencias de elementos (what?)

$$\text{aux } sumDist(a, b : [\text{Int}]) : [\text{Int}] = [x + y | x \leftarrow a, y \leftarrow b, x \neq y];$$

Ejemplo:

$$sumDist([1, 2, 3], [2, 3, 4, 5]) == [3, 4, 5, 6, 5, 6, 7, 5, 7, 8]$$

Operaciones con secuencias

- Longitud: $long(a : [T]) : \text{Int}$
  - Retorna la longitud de la secuencia a
  - Notación: $long(a)$ se puede escribir $|a|$
- Indexación: $indice(a : [T], i : \text{Int}) : T$
  - Requiere $0 \leq i < |a|$;
  - Retorna el elemento en la i-ésima posición de a
  - La primera posición es la 0
  - Notación: $indice(a, i)$ se puede escribir $a[i]$ y también a_i
- Cabeza: $cab(a : [T]) : T$
  - Requiere $|a| > 0$;
  - Retorna el primer elemento de la secuencia

Más operaciones

- Cola: $cola(a : [T]) : [T]$
  - Requiere $|a| > 0$;
  - Retorna la secuencia sin su primer elemento
- Pertenencia: $en(t : T, a : [T]) : \text{Bool}$
  - Indica si el elemento aparece (al menos una vez) en la secuencia
  - Notación: $en(t, a)$ se puede escribir t en a y también $t \in a$
  - $t \notin a$ es $\neg(t \in a)$
- Agregar cabeza: $cons(t : T, a : [T]) : [T]$
  - Retorna una secuencia como a, agregándole t como primer elemento
  - Notación: $cons(t, a)$ se puede escribir $t : a$

Más operaciones

- Concatenación: $conc(a, b : [T]) : [T]$
  - Retorna una secuencia con los elementos de a, seguidos de los de b
  - Notación: $conc(a, b)$ se puede escribir $a + +b$
- Subsecuencia: $sub(a : [T], d, h : \text{Int}) : [T]$
  - Returna la sublista de a en las posiciones entre d y h (ambas inclusive)
  - Cuando no es $0 \leq d \leq h < |a|$, da la secuencia vacía
- Asignación a una posición: $cambiar(a : [T], i : \text{Int}, val : T) : [T]$
  - Requiere $0 \leq i < |a|$;
  - Retorna una secuencia igual a a, pero el valor en la posición i es val

Subsecuencias con intervalos

- Notación para obtener la subsecuencia de una secuencia dada que está en un intervalo de posiciones
- Admite intervalos abiertos
  - $a[d..h] == sub(a, d, h)$
  - $a[d..h) == sub(a, d, h-1)$
  - $a(d..h] == sub(a, d+1, h)$
  - $a(d..h) == sub(a, d+1, h-1)$
  - $a[d..] == sub(a, d, |a|-1)$
  - $a(d..] == sub(a, d+1, |a|-1)$

Operaciones numéricas de combinación

- Sumatoria: $sum(sec : [T]) : T$
  - T debe ser un tipo numérico (Float, Int)
  - Calcula la suma de todos los elementos de la secuencia
  - Si sec es vacía, el resultado es 0
  - Notación: $sum(sec)$ se puede escribir $\sum sec$
  - Ejemplo:
    aux $potNegDosHasta(n : \text{Int}) : \text{Float} = \sum[2^{-m} | m \leftarrow [1..n]]$;
- Productoria: $prod(sec : [T]) : T$
  - T debe ser un tipo numérico (Float, Int)
  - Calcula el producto de todos los elementos de la secuencia
  - Si sec es vacía, el resultado es 1
  - Notación: $prod(sec)$ se puede escribir $\prod sec$

Operaciones booleanas de combinación

- Todos verdaderos: $todos(sec : [\text{Bool}]) : \text{Bool}$
  - Es verdadero solamente si todos los elementos de la secuencia son True (o la secuencia es vacía)
- Alguno verdadero: $alguno(sec : [\text{Bool}]) : \text{Bool}$
  - Es verdadero solamente si algún elemento de la secuencia es True (y ninguno está indefinido)

Para todo, variante notacional de todos( )

(∀ *selectores*, *condiciones*) *expresión*

- Término de tipo Bool, que afirma que todos los elementos de una lista por comprensión cumplen una propiedad
- Equivale a escribir *todos*([*expresión* | *selectores*, *condiciones*])
- Ejemplo: "todos los elementos en posiciones pares son > 5":

  aux $par(n : \text{Int}) : \text{Bool} = n \bmod 2 == 0$;

  aux $posParM5(a : [\text{Float}]) : \text{Bool} = (\forall i \leftarrow [0..|a|), par(i))\ a[i] > 5$;

- Esta expresión es equivalente a ...

  $todos([a[i] > 5 | i \leftarrow [0..|a|),\ par(i)])$;

Existe, variante notacional de alguno( )

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

- Informa si algún elemento cumple la propiedad
- Equivale a *alguno*([*expresión* | *selectores*, *condiciones*])
- **Notación:** en lugar de ∃ se puede escribir existe o existen
- Ejemplo: "Hay algún elemento de la lista que es par y mayor que 5":

  aux $hayParM5(a : [\text{Int}]) : \text{Bool} = (\exists x \leftarrow a, par(x))\ x > 5$;

- Es equivalente a ...

  $alguno([x > 5 | x \leftarrow a, par(x)])$;

Cantidades

- Es habitual contar cuántos elementos de una secuencia cumplen una condición
- Para eso, medimos la longitud de una secuencia definida por comprensión
- Ejemplo: "¿cuántas veces aparece el elemento x en la secuencia a?"

  aux $cuenta(x : T, a : [T]) : \text{Int} = long([y | y \leftarrow a, y == x])$;

- Podemos usar esta función para saber si dos secuencias tienen los mismos elementos (en otro orden):

  aux $mismos(a, b : [T]) : \text{Bool} = (|a| == |b| \wedge$
  $(\forall c \leftarrow a)\ cuenta(c, a) == cuenta(c, b))$;

Cantidades

- Otro ejemplo: "¿cuántos primos positivos menores a n existen?"

  aux $primosMenores(n : \text{Int}) : \text{Int} =$
  $long([y \mid y \leftarrow [0..n), primo(y)])$;

  aux $primo(n : \text{Int}) : \text{Bool} =$
  $(n \geq 2 \wedge \neg(\exists m \leftarrow [2..n))\ n \bmod m == 0)$;

Acumulación

$$acum(expresión \mid inicializacion, selectores, condición)$$

- Construye un valor a partir de una o más secuencias
- *inicializacion* tiene la forma *acumulador* : *tipoAcum* = *init*
  - *acumulador* es un nombre de variable (nuevo)
  - *init* es una expresión de tipo *tipoAcum*
- El *acumulador* puede aparecer en la *expresión*, pero no en la *condición*
- Significado:
  - El valor inicial de *acumulador* es el valor de *init*
  - Por cada valor de los *selectores*, se calcula la *expresión*
  - Ese es el nuevo valor que toma el *acumulador*
  - El resultado de *acum* es el resultado final del *acumulador*

Ejemplos de acumulación

- Sumatoria y productoria:

  aux $sum(l : [\text{Float}]) : \text{Float} = acum(s + i \mid s : \text{Float} = 0, i \leftarrow l)$;

  aux $prod(l : [\text{Float}]) : \text{Float} = acum(p * i \mid p : \text{Float} = 1, i \leftarrow l)$;

- Secuencia de números de Fibonacci:

  aux $fiboSuc(n : \text{Int}) : [\text{Int}] =$
  $acum(f + +[f[i-1] + f[i-2]] \mid f : [\text{Int}] = [1,1], i \leftarrow [2..n])$;

  - si $n \geq 1$, devuelve los primeros $n+1$ números de Fibonacci
  - si $n == 0$, devuelve $[1,1]$

- n-ésimo número de Fibonacci ($n \geq 1$)

  aux $fibo(n : \text{Int}) : \text{Int} = (fiboSuc(n-1))[n-1]$;

Más ejemplos de acumulación

- Concatenación de elementos de una secuencia de secuencias (aplanar)

  aux $concat(a : [[T]]) : [T] =$
  $acum(l + +c \mid l : [T] = [], c \leftarrow a)$;

- Por ejemplo, $concat([[1,2],[3],[4,5,6]]) == [1,2,3,4,5,6]$

- El término [*expresión* | *selectores*, *condición*], correspondiente a la definición de secuencias por comprensión, es equivalente a:

  $acum(res + +[expresión] \mid res : [T] = [], selectores, condición)$;

Especificación de problemas

1. Tenemos un **problema** a resolver por medio de una computadora ...
2. para el cual escribimos una **especificación**, que es el contrato que debe cumplir un programa para ser considerado solución del problema en cuestión.
3. Sobre la base de la especificación, pensaremos un **algoritmo** para la especificación ...
4. ... y lo implementaremos, escribiendo un **programa** en algún lenguaje de programación adecuado.

Ejemplos de especificación

- Calcular el cociente de dos enteros

problema división(a, b : Int) = *result* : Int {
  requiere $b \neq 0$;
  asegura *result* == a div b;
}

- Calcular el cociente y el resto, para divisor positivo
  - Necesitamos devolver dos valores
  - Usamos el tipo (Int, Int) son los pares ordenados de enteros *prm* y *sgd* devuelven sus componentes

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

Parámetros modificables

- Alternativa 2:
  - Único resultado: el cociente
  - Resto: parámetro modificable

problema cocienteResto2(a, b, r : Int) = q : Int {
  requiere $b > 0$;
  modifica r;
  asegura $a == q * b + r \wedge 0 \leq r < b$;
}

- Alternativa 3:
  - Otro parámetro para el cociente
  - La función no tiene resultado

problema cocienteResto3(a, b, q, r : Int){
  requiere $b > 0$;
  modifica q, r;
  asegura $a == q * b + r \wedge 0 \leq r < b$;
}

Parámetros modificables. pre()

Problema: Incrementar en uno el valor de entrada

- Alternativa 1: Usamos una función ...
problema incremento1(a : Int) = *res* : *Int*{
  asegura *res* == $a + 1$;
}
- Alternativa 2: Usamos el mismo argumento para la entrada y para la salida ...

problema incremento2(a : Int){
  modifica a;
  asegura $a == pre(a) + 1$;
}

Más ejemplos de especificación

Sumar los inversos multiplicativos de varios reales

- Como no sabemos la cantidad, usamos secuencias

problema sumarInvertidos(a : [Float]) = *result* : Float {
  requiere $0 \notin a$;
  asegura *result* = $\sum[1/x \mid x \leftarrow a]$;
}

- **Precondición:** que el argumento no contenga ningún 0. Si no, la poscondición podría indefinirse
- Formas equivalentes:
  - requiere $(\forall x \leftarrow a)\ x \neq 0$;
  - requiere $\neg(\exists x \leftarrow a)\ x == 0$;
  - requiere $\neg(\exists i \leftarrow [0..|a|))\ a[i] == 0$;

Más ejemplos de especificación

Encontrar una raíz de un polinomio de grado 2 a coeficientes reales

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

- No tiene precondición (la precondición es True). Pero, por ejemplo, si $a == 1$, $b == 0$ y $c == 1$, no existe ningún r tal que $r * r + 1 == 0$
- Especificación que no puede ser cumplida por ninguna función. La precondición es demasiado débil
- La nueva precondición podría ser:
  requiere $b * b \geq 4 * a * c$;
- La postcondición describe qué hacer y no cómo hacerlo
  No dice cómo calcular la raíz, ni qué raíz devolver
- Equivalentemente podríamos poner
  asegura $r == (-b + (b^2 - 4 * a * c)^{1/2})/(2 * a)||$
  $r == (-b - (b^2 - 4 * a * c)^{1/2})/(2 * a)$
  No la preferimos porque induce una forma de calcular la solución

Otro ejemplo

problema índiceMenorDistintos(a : [Float]) = res : Int {
  requiere NoNegativos: $(\forall x \leftarrow a)\ x \geq 0$;
  requiere Distintos: $(\forall i \leftarrow [0..|a|), j \leftarrow [0..|a|), i \neq j)\ a_i \neq a_j$;
  asegura $0 \leq res < |a|$;
  asegura $(\forall x \leftarrow a)\ a[res] \leq x$;
}

- Nombramos las precondiciones, para aclarar su significado
  - Los nombres también pueden usarse como predicados en cualquier lugar de la especificación
  - El nombre no alcanza, hay que escribir la precondición en el lenguaje
- Otra forma de escribir la segunda precondición:
  requiere Distintos2: $(\forall i \leftarrow [0..|a|))\ a[i] \notin a[0..i)$;

Sobre-especificación

Se llama así a la situación en la cual la postcondición impone más restricciones que las necesarias. Limita excesivamente los posibles resultados de la especificación.

Problema: dada una lista, retornar una lista con los mismos elementos que no esté ordenada de menor a mayor.

problema Desordena(a : [Int]) = res : [Int] {
  asegura mismos(a, res);
  asegura $(\forall i \leftarrow [0..|a| - 1))\ a[i] \geq a[i + 1]]$;
}
Está sobre-especificando porque exige el orden de mayor a menor.

problema Desordena(a : [Int]) = res : [Int] {
  asegura mismos(a, res);
  asegura $(\exists i \leftarrow [0..|a| - 1))\ a[i] > a[i + 1]]$;
}

Otro ejemplo de sobre-especificación

¡Cuidado! es bastante fácil caer involuntariamente en la sobrespecificación cuando usamos listas por comprensión.

Problema: dar los índices de los numeros pares de una lista, en cualquier orden.

problema índicesDePares(a : [Int]) = res : [Int]{
  asegura $res == [i | i \leftarrow [0..|a|), a[i] mod 2 == 0]$;
}
... sobre-especifica el problema, porque exige que res siga el orden de a.

problema índicesDePares(a : [Int]) = res : [Int]{
  asegura $0 \leq |res| \leq |a|$;
  asegura mismos($res, [i | i \leftarrow [0..|a|), a[i] mod 2 == 0]$);
}