Algoritmos y Estructuras de Datos I

Departamento de Computación, FCEyN, UBA

2° cuatrimestre de 2008

Contenido

Especificación

Clase 1

Introducción a la especificación de problemas

Cursada

- clases teóricas
  - Paula Zabala y Santiago Figueira
- clases prácticas
  - Carlos López Pombo, Pablo Turjanski y Martín Urtasun
- sitio web de la materia: www.dc.uba.ar/people/materias/algo1
- régimen de aprobación
  - parciales
    - 3 parciales
    - 3 recuperatorios (al final de la cursada)
  - trabajos prácticos
    - 3 entregas
    - 3 recuperatorios (cada uno a continuación)
    - grupos de 4 alumnos
  - examen final (si lo dan en diciembre 2008 o febrero/marzo 2009, cuenta la nota de la cursada)

Objetivos y contenidos

- Objetivos:
  - especificar problemas
    - describirlos de forma no ambigua
  - escribir programas sencillos
    - tratamiento de secuencias
  - razonar acerca de estos programas
    - demostrar matemáticamente que un programa es correcto
    - vision abstracta del proceso de computación
    - manejo simbólico y herramientas para demostrar
- Contenidos:
  - especificación
    - describir problemas en un lenguaje formal (preciso, claro, abstracto)
  - programación funcional (Haskell)
    - parecido al lenguaje matemático
    - escribir de manera simple algoritmos y estructuras de datos
  - programación imperativa (C++)
    - paradigma más difundido
    - más eficiente
    - se necesita para seguir la carrera

Especificación, algoritmo, programa

1. especificación = descripción del problema
   - ¿qué problema tenemos?
   - en lenguaje formal
   - describe propiedades de la solución
   - Dijkstra, Hoare (años 70)
2. algoritmo = descripción de la solución (escrito para humanos)
   - ¿cómo resolvemos el problema?
3. programa = descripción de la solución (escrito para la computadora)
   - también, ¿cómo resolvemos el problema?
   - usando un lenguaje de programación

Problemas y solución

Si voy a escribir un programa, es porque hay un problema a resolver

- a veces, la descripción es vaga o ambigua
- no siempre es claro que haya solución
- no siempre involucra uno o más programas de computación

Ejemplos de problemas:

- cerré el auto con las llaves adentro
  - seguramente tiene solución, sin programas
- quiero calcular la edad de una persona
  - tal vez podamos crear un programa
- necesitamos que la empresa reduzca sus gastos en un 10 %
  - puede o no tener solución, que puede o no requerir programas nuevos o existentes
- soy muy petiso
  - quizás no tenga solución

Problema vs. solución

Parece una diferencia trivial

- en los ejemplos, es clara
- pero es fácil confundirlos

Ejemplo:

- confundir el problema con la solución:
  A: “tengo un problema: necesito un alambre”
  B: “no tengo”
- en realidad, ese no era el problema:
  A: “tengo un problema: cerré el auto con las llaves adentro”
  B: “tomá la copia de la llave que me pediste que te guardara cuando compraste el auto”
- confundir el problema con la solución trae nuevos problemas
  - es muy común en computación
- los programas de computadora casi nunca son la solución completa
  - a veces forman parte de la solución
  - ¡y a veces ni siquiera se necesitan!

Etapas en el desarrollo de programas

1. especificación: definición precisa del problema

↓

2. diseño: elegir una solución y dividir el problema en partes

↓

3. programación: escribir algoritmos e implementarlos en algún lenguaje

↓

4. validación: ver si el programa cumple con lo especificado

↓

5. mantenimiento: corregir errores y adaptarlo a nuevos requerimientos

E	D	P	V	M	···

1. Especificación

- el planteo inicial del problema es vago y ambiguo
- especificación = descripción clara y precisa
- óptimo: lenguaje formal, por ejemplo, el lenguaje de la lógica matemática

Por ejemplo, el problema de calcular de edad de una persona parece bien planteado, pero:

- ¿cómo recibo los datos? manualmente, archivo en disco, sensor
- ¿cuáles son los datos? fecha de nacimiento, tamaño de la persona, deterioro celular
- ¿cómo devuelvo el resultado? pantalla, papel, voz alta, email
- ¿qué forma van a tener? días, años enteros, fracción de años

Una buena especificación responde algunas:

- *Necesito una función que, dadas dos fechas en formato dd/mm/aaaa, me devuelva la cantidad de días que hay entre ellas.*

todavía faltan los requerimientos no funcionales

- forma de entrada de parámetros y de salida de resultados, tipo de computadora, tiempo con el que se cuenta para programarlo, etc.
- no son parte de la especificación funcional y quedan para otras materias

2. Diseño

Etapa en la que se responde

- ¿varios programas o uno muy complejo?
  - ¿cómo dividirlo en partes, qué porción del problema resuelve cada una?
  - ¿distintas partes en distintas máquinas?
  - estudio de las partes que lo componen
    - (mini) especificación de cada parte
    - un programador recibe una sola (o una por vez)
- ¿programas ya hechos con los que interactuar?
- lo van a ver en *Algoritmos y Estructuras de Datos II* y en *Ingeniería del Software I*

3. Programación - Algoritmos

Pensar en la solución al problema. Escribir un algoritmo:

- pasos precisos para llegar al resultado buscado de manera efectiva
- primero tienen que estar definidos los pasos primitivos

Ejemplo de algoritmo para la especificación

> *Necesito una función que, dadas dos fechas a y b en formato dd/mm/aaaa, me devuelva la cantidad de días que hay entre ellas.*

1. restar el año de *b* al año de *a*
2. multiplicar el resultado por 365
3. sumarle la cantidad de días desde el 1° de enero del año de *b* hasta el día *b*
4. restarle la cantidad de días desde el 1° de enero del año de *a* hasta el día *a*
5. sumarle la cantidad de 29 de febrero que hubo en el período
6. devolver ese resultado, acompañado de la palabra "días"

¿Son todos primitivos? (3, 4, 5)

- si no, hay que dar algoritmos para realizarlos

Pasos primitivos

- problema: sumar dos números naturales
- algoritmos:
  - voy sumando uno al primero y restando uno al segundo, hasta que llegue a cero
  - sumo las unidades del primero a las del segundo, después las decenas y así ("llevándome uno" cuando hace falta)
  - escribo el primero en una calculadora, aprieto $+$, escribo el segundo, aprieto $=$
- los tres son válidos
- depende de qué operaciones primitivas tenga
  - sumar / restar uno
  - sumar dígitos (y concatenar resultados)
  - apretar una tecla de una calculadora

3. Programación - Programas

- traducir el algoritmo (escrito o idea) para que una computadora lo entienda
- lenguaje de programación
  - vamos a empezar usando uno: Haskell
  - después, C++
- hay muchos otros
  - pueden ser más adecuados para ciertas tareas
  - depende del algoritmo, de la interfaz, tiempo de ejecución, tipo de máquina, interoperabilidad, entrenamiento de los programadores, licencias, etc.

Representación de los datos

- ya vimos que era importante en el ejemplo de calcular la edad de una persona (días, meses, años, fracciones de año)
- otro ejemplo:
  - contar la cantidad de alumnos de cada sexo
  - yo puedo darme cuenta a simple vista (a veces)
  - la computadora probablemente no. ¿Que dato conviene tener?
    - foto
    - foto del documento
    - ADN
    - valor de la propiedad *sexo* para cada alumno
- estructuras de datos
  - necesarias para especificar
  - puede ser que haya que cambiarlas al programar
  - las van a estudiar a fondo en *Algoritmos y Estructuras de Datos II*

4. Validación

Asegurarse de que un programa cumple con la especificación

- testing
  - probar el programa con muchos datos y ver si hace lo que tiene que hacer
  - en general, para estar seguro de que anda bien, uno tendría que probarlo para infinitos datos de entrada
  - si hay un error, con algo de suerte uno puede encontrarlo
  - no es infalible (puede pasar el testing pero haber errores)
  - en *Ingeniería del Software I* van a ver técnicas para hacer testing
- verificación formal
  - demostrar matemáticamente que un programa cumple con una especificación
  - es más difícil que hacer testing
  - una demostración cubre infinitos valores de entrada a la vez (abstracción)
  - es infalible (si está demostrado, el programa no tiene errores)
  - en esta materia van a estudiar cómo demostrar que programas simples son correctos para una especificación

5. Mantenimiento

- tiempo después, encontramos errores
  - el programa no cumplía la especificación
  - la especificación no describía correctamente el problema
- o cambian los requerimientos
- puede hacerlo el mismo equipo u otro
- justifica las etapas anteriores
  - si se hicieron bien la especificación, diseño, programación y validación, las modificaciones van a ser más sencillas y menos frecuentes

Especificación

- objetivos:
  - antes de programar: entender el problema
  - después de programar: determinar si el programa es correcto
    - testing
    - verificación formal
    - derivación (construyo el programa a partir de la especificación)
- estrategia:
  - evitar pensar (todavía) una solución para el problema
  - limitarse a describir cuál es el problema a resolver
    - qué propiedades tiene que cumplir una solución para resolver el problema
    - buscamos el qué y no el cómo

Instancias de un problema

- ejemplo de problema: arreglar una cafetera
- para solucionarlo, necesitamos más datos
  - ¿qué clase de cafetera es?
  - ¿qué defecto tiene?
  - ¿cuánto presupuesto tenemos?
- se llaman parámetros del problema
- cada combinación de valores de los parámetros es una instancia del problema
  - una instancia: "arreglar una cafetera de filtro cuya jarra pierde agua, gastando a lo sumo $30"
- los valores de los parámetros se llaman argumentos

Problemas funcionales

- no podemos especificar formalmente cualquier problema
  - el hambre en el mundo
  - la salud de Sandro
- simplificación (para esta materia)
  - problemas que puedan solucionarse con una función
    - parámetros de entrada
    - un resultado para cada combinación de valores de entrada

Tipos de datos

Cada parámetro tiene un tipo de datos

- conjunto de valores para los que hay ciertas operaciones definidas

Por ejemplo:

- parámetros de tipo *fecha*
  - valores: ternas de números enteros
  - operaciones: comparación, obtener el año,...
- parámetros de tipo *dinero*
  - valores: números reales con dos decimales
  - operaciones: suma, resta,...

Encabezado de un problema

Indica la forma que debe tener una solución.

problema *nombre*(*parámetros*) = *nombreRes* : *tipoRes*

- *nombre*: nombre que le damos al problema
  - será resuelto por una función con ese mismo nombre
- *nombreRes*: nombre que le damos al resultado
- *tipoRes*: tipo de datos del resultado
- *parámetros*: lista que da el tipo y el nombre de cada uno

Ejemplo

- problema *resta*(*minuendo*, *sustraendo* : Int) = *res* : Int
- la función se llama resta
- da un resultado que vamos a llamar res
  - y es de tipo Int (los enteros)
- depende de dos parámetros: minuendo y sustraendo
  - también son de tipo Int

Contratos

- la función que solucione el problema va a ser llamada o invocada por un usuario
  - puede ser el programador de otra función
- especificación = contrato entre el programador de una función que resuelva el problema y el usuario de esa función

Por ejemplo:

- problema: calcular la raíz cuadrada de un real
- solución (función)
  - va a tener un parámetro real
  - va a calcular un resultado real
- para hacer el cálculo, debe recibir un número no negativo
  - compromiso para el usuario: no puede proveer números negativos
  - derecho para el programador de la función: puede suponer que el argumento recibido no es negativo
- el resultado va a ser la raíz del número
  - compromiso 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

Partes de una especificación

Tiene 3 partes

1. encabezado (ya lo vimos)
2. precondición
   - condición sobre los argumentos
   - el programador da por cierta
   - lo que requiere la función para hacer su tarea
   - por ejemplo: "el valor de entrada es un real no negativo"
3. poscondición
   - condición sobre el resultado
   - debe ser cumplida por el programador, siempre y cuando el usuario haya cumplido la precondición
   - 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 del valor de entrada"

El contrato dice:

El programador va a hacer un programa P tal que si el usuario suministra datos que hacen verdadera la precondición, entonces P va a terminar en una cantidad finita de pasos y va a devolver un valor que hace verdadera la poscondición.

- si el usuario no cumple la precondición y P se cuelga o no cumple la poscondición...
  - ¿el usuario tiene derecho a quejarse? No.
  - ¿se viola el contrato? No. El contrato prevé este caso y dice que P solo debe funcionar en caso de que el usuario cumpla con la precondición
- si el usuario cumple la precondición y P se cuelga o no cumple la poscondición...
  - ¿el usuario tiene derecho a quejarse? Sí.
  - ¿se viola el contrato? Sí. Es el único caso en que se viola el contrato. El programador no hizo lo que se había comprometido a hacer.

Lenguaje naturales y lenguajes formales

- lenguajes naturales
  - idiomas (castellano)
  - mucho poder expresivo (emociones, deseos, suposiciones y demás)
  - con un costo (conocimiento del contexto, experiencias compartidas ambigüedad e imprecisión)
  - queremos evitarlo al especificar
- lenguajes formales
  - limitan lo que se puede expresar
  - todas las suposiciones quedan explícitas
  - relación directa entre lo escrito (sintaxis) y su significado (semántica)
  - pueden tratarse formalmente
    - símbolos manipulables directamente
    - seguridad de que las manipulaciones son válidas también para el significado
  - ejemplo: aritmética
    - lenguaje formal para los números y sus operaciones
    - resolvemos problemas simbólicamente sin pensar en los significados numéricos y llegamos a resultados correctos
  - en *Teoría de Lenguajes* y *Lógica y Computabilidad* van a estudiarlos en profundidad

Especificación formal

- ya aprendimos a escribir encabezados
- tenemos que dar también
  - la precondición (lo que la función requiere)
  - la poscondición (lo que asegura)
- usamos un lenguaje formal para describir la precondición y la poscondición
- ejemplo de problema: calcular la raíz cuadrada de un número
  - lo llamamos rcuad
  - Float representa el conjunto $\mathbb{R}$

```
problema rcuad(x : Float) = result : Float {
  requiere x ≥ 0;
  asegura result * result == x;
}
```


Especificación

Clase 2

Lógica proposicional - tipos básicos

Lógica proposicional - sintaxis

- símbolos

$$\text{true} \ , \ \text{false} \ , \ \bot \ , \ \neg \ , \ \wedge \ , \ \vee \ , \ \rightarrow \ , \ \leftrightarrow \ , \ (\ , \ )$$

- variables proposicionales (infinitas)

$$p \ , \ q \ , \ r \ , \ \ldots$$

- fórmulas
  1. true, false y $\bot$ son fórmulas
  2. cualquier variable proposicional es una fórmula
  3. si A es una fórmula, $\neg A$ es una fórmula
  4. si $A_1, A_2, \ldots, A_n$ son fórmulas, $(A_1 \wedge A_2 \wedge \cdots \wedge A_n)$ es una fórmula
  5. si $A_1, A_2, \ldots, A_n$ son fórmulas, $(A_1 \vee A_2 \vee \cdots \vee A_n)$ es una fórmula
  6. si A y B son fórmulas, $(A \rightarrow B)$ es una fórmula
  7. si A y B son fórmulas, $(A \leftrightarrow B)$ es una fórmula

Ejemplos

¿Cuáles son fórmulas?

- $p \vee q$ no
- $(p \vee q)$ sí
- $p \vee q \rightarrow r$ no
- $(p \vee q) \rightarrow r$ no
- $((p \vee q) \rightarrow r)$ sí
- $(p \rightarrow q \rightarrow r)$ no

Semántica clásica

- 2 valores de verdad posibles
  1. verdadero (1)
  2. falso (0)
- interpretación:
  - true siempre vale 1
  - false siempre vale 0
  - $\neg$ se interpreta como "no", se llama negación
  - $\wedge$ se interpreta como "y", se llama conjunción
  - $\vee$ se interpreta como "o"(no exclusivo), se llama disyunción
  - $\rightarrow$ se interpreta como "si... entonces", se llama implicación
  - $\leftrightarrow$ se interpreta como "si y solo si", se llama doble implicación o equivalencia

Semántica clásica

Conociendo el valor de las variables proposicionales de una fórmula, conocemos el valor de verdad de la fórmula

p	$\neg p$
1	0
0	1

p	q	$(p \wedge q)$
1	1	1
1	0	0
0	1	0
0	0	0

p	q	$(p \vee q)$
1	1	1
1	0	1
0	1	1
0	0	0

p	q	$(p \rightarrow q)$
1	1	1
1	0	0
0	1	1
0	0	1

p	q	$(p \leftrightarrow q)$
1	1	1
1	0	0
0	1	0
0	0	1

Ejemplo: tabla de verdad para $((p \wedge q) \rightarrow r)$

p	q	r	$(p \wedge q)$	$((p \wedge q) \rightarrow r)$
1	1	1	1	1
1	1	0	1	0
1	0	1	0	1
1	0	0	0	1
0	1	1	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	0	1

Semántica trivaluada

- 3 valores de verdad posibles
  1. verdadero (1)
  2. falso (0)
  3. indefinido ($-$)
- es la que vamos a usar en esta materia
- ¿por qué?
  - queremos especificar problemas que puedan resolverse con un algoritmo
  - puede ser que un algoritmo realice una operación inválida
    - dividir por cero
    - raíz cuadrada de un número negativo
  - necesitamos contemplar esta posibilidad en la especificación
- interpretación:
  - true siempre vale 1
  - false siempre vale 0
  - $\perp$ siempre vale $-$
  - se extienden las definiciones de $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Semántica trivaluada (secuencial)

Se llama secuencial porque

- los términos se evalúan de izquierda a derecha
- la evaluación termina cuando se puede deducir el valor de verdad, aunque el resto esté indefinido

Extendemos la semántica de $\neg$, $\wedge$, $\vee$

p	$\neg p$
1	0
0	1
$-$	$-$

p	q	$(p \wedge q)$
1	1	1
1	0	0
0	1	0
0	0	0
1	$-$	$-$
0	$-$	0
$-$	1	$-$
$-$	0	$-$
$-$	$-$	$-$

p	q	$(p \vee q)$
1	1	1
1	0	1
0	1	1
0	0	0
1	$-$	1
0	$-$	$-$
$-$	1	$-$
$-$	0	$-$
$-$	$-$	$-$

Semántica trivaluada (secuencial)

Extendemos la semántica de $\rightarrow$, $\leftrightarrow$

p	q	$(p \rightarrow q)$
1	1	1
1	0	0
0	1	1
0	0	1
1	$-$	$-$
0	$-$	1
$-$	1	$-$
$-$	0	$-$
$-$	$-$	$-$

p	q	$(p \leftrightarrow q)$
1	1	1
1	0	0
0	1	0
0	0	1
1	$-$	$-$
0	$-$	$-$
$-$	1	$-$
$-$	0	$-$
$-$	$-$	$-$

Dos conectivos bastan

- $\neg$ y $\vee$
  - $(A \wedge B)$ es $\neg(\neg A \vee \neg B)$
  - $(A \rightarrow B)$ es $(\neg A \vee B)$
  - true es $(A \vee \neg A)$
  - false es $\neg$true
- $\neg$ y $\wedge$
  - $(A \vee B)$ es $\neg(\neg A \wedge \neg B)$
  - $(A \rightarrow B)$ es $\neg(A \wedge \neg B)$
  - false es $(A \wedge \neg A)$
  - true es $\neg$false
- $\neg$ y $\rightarrow$
  - $(A \vee B)$ es $(\neg A \rightarrow B)$
  - $(A \wedge B)$ es $\neg(A \rightarrow \neg B)$
  - true es $(A \rightarrow A)$
  - false es $\neg$true

Tautologías, contradicciones y contingencias

- una fórmula es una tautología si siempre toma el valor 1 para valores definidos de sus variables proposicionales
  Por ejemplo, $((p \wedge q) \rightarrow p)$ es tautología:

p	q	$(p \wedge q)$	$((p \wedge q) \rightarrow p)$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

- una fórmula es una contradicción si siempre toma el valor 0 para valores definidos de sus variables proposicionales
  Por ejemplo, $(p \wedge \neg p)$ es contradicción:

p	$\neg p$	$(p \wedge \neg p)$
1	0	0
0	1	0

- una fórmula es una contingencia cuando no es ni tautología ni contradicción

Relación de fuerza

Decimos que A es más fuerte que B cuando $(A \rightarrow B)$ es tautología.

También decimos que A fuerza a B o que B es más débil que A.

Por ejemplo,

- ¿$(p \wedge q)$ es más fuerte que p? sí
- ¿$(p \vee q)$ es más fuerte que p? no
- ¿p es más fuerte que $(q \rightarrow p)$? sí
- ¿p es más fuerte que q? no
- ¿p es más fuerte que p? sí
- ¿hay una fórmula más fuerte que todas? sí, por ej. false
- ¿hay una fórmula más débil que todas? sí, por ej. true

Lenguaje de especificación

- hasta ahora vimos lógica proposicional
  - es muy limitada
- nuestro objetivo es especificar (describir problemas)
- vamos a usar un lenguaje más poderoso
  - permite hablar de elementos y sus propiedades
  - es un lenguaje tipado
    - los elementos pertenecen a distintos dominios o conjuntos (enteros, reales, etc.)

Tipos de datos

- conjunto de valores con operaciones
- vamos a empezar viendo tipos básicos
- para hablar de un elemento de un tipo T en nuestro lenguaje, escribimos un término o expresión
  - variable de tipo T
  - constante de tipo T
  - función (operación) aplicada a otros términos (del tipo T o de otro tipo)
- todos los tipos tienen un elemento distinguido: $\perp$ o Indef

Tipo Bool (valor de verdad)

- valores: 1, 0 , $-$
- constantes: true, false, $\perp$ (o Indef)
- conectivos lógicos: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ con la semántica trivaluada que vimos antes
  - $\neg A$ se puede escribir no(A)
  - $(A \wedge B)$ se puede escribir (A && B)
  - $(A \vee B)$ se puede escribir (A || B)
  - $(A \rightarrow B)$ se puede escribir (A --> B)
  - $(A \leftrightarrow B)$ se puede escribir (A <--> B)
- comparación: $A == B$
  - todos los tipos tienen esta operación (A y B deben ser del mismo tipo T)
  - es de tipo bool
  - es verdadero si el valor de A igual al valor de B (salvo que alguno esté indefinido - ver hoja 47)
  - $A \neq B$ o A != B es equivalente a $\neg(A == B)$
- semántica secuencial

Tipo Int (números enteros)

- sus elementos son los de $\mathbb{Z}$
- constantes: 0 ; 1 ; -1 ; 2 ; -2 ; ...
- operaciones aritméticas
  - $a + b$ (suma)
  - $a - b$ (resta)
  - $a * b$ (multiplicación)
  - a div b (división entera)
  - a mod b (resto de dividir a a por b)
  - a^b o pot(a,b) (potencia)
  - abs(a) (valor absoluto)
- comparaciones (de tipo bool)
  - $a < b$ (menor)
  - $a \leq b$ o a <= b (menor o igual)
  - $a > b$ (mayor)
  - $a \geq b$ o a >= b (mayor o igual)
- β o beta. Si A es de tipo Bool, se definide como:

$$\beta(A) = \text{beta}(A) = \begin{cases} 1 & \text{si } A \text{ es verdadero} \\ 0 & \text{si } A \text{ es falso} \\ - & \text{si } A \text{ es indefinido} \end{cases}$$

Tipo Float (números reales)

- sus elementos son los de $\mathbb{R}$
- constantes: 0 ; 1 ; -7 ; 81 ; 7,4552 ; π ...
- operaciones aritméticas
  - las mismas que Int, salvo div y mod
  - a/b (división)
  - $\log_b(a)$ (logaritmo)
  - trigonométricas
- comparaciones (de tipo bool)
  - las mismas que para Int
- conversión a entero
  - $\lfloor a \rfloor$ o int(a)
- todos los términos de tipo Int pueden usarse como términos de tipo Float

Tipo Char (caracteres)

- sus elementos son los las letras, dígitos y símbolos
- constantes:
  $`a`, `b`, `c`, \ldots, `z`, \ldots, `A`, `B`, `C`, \ldots, `Z`, \ldots, `0`, `1`, `2`, \ldots, `9`$
  (en algún orden)
- función ord
  - numera todos los caracteres
  - no importa mucho cuál es el valor de cada uno, pero
    - $\text{ord}(`a`) + 1 == \text{ord}(`b`)$
    - $\text{ord}(`A`) + 1 == \text{ord}(`B`)$
    - $\text{ord}(`1`) + 1 == \text{ord}(`2`)$
- función char
  - es la inversa de ord
- las comparaciones entre caracteres son comparaciones entre sus órdenes
  - $a < b$ es equivalente a $\text{ord}(a) < \text{ord}(b)$
  - lo mismo para $\leq$, $>$, $\geq$

Términos

- simples
  - variables del tipo o
  - constantes del tipo
- complejos
  - combinaciones de funciones aplicadas a funciones, constantes y variables

Ejemplos de términos de tipo Int

- $0 + 1$
- $((3 + 4) * 7)^2 - 1$
- $2 * \beta(1 + 1 == 2)$
- $1 + \text{ord}(`A`)$
- con x variable de tipo Int; y de tipo Float; z de tipo Bool
  - $2 * x + 1$
  - $\beta(y^2 > \pi) + x$
  - $(x \bmod 3) * \beta(z)$

Semántica de los términos

- vimos que los términos representan elementos de los tipos
- los términos tienen valor indefinido cuando no se puede hacer alguna operación
  - 1 div 0
  - $(-1)^{1/2}$
- las operaciones son estrictas (salvo los conectivos de bool)
  - si uno de sus argumentos es indefinido, el resultado también está indefinido
  - ejemplos
    - $0 * (-1)^{1/2}$ es indefinido ($*$ es estricto)
    - $0^{1/0}$ es indefinido (pot es estricto)
    - $((1 + 1 == 2) \vee (0 > 1/0))$ es verdadero ($\vee$ no es estricto)
  - las comparaciones con $\perp$ son indefinidas
    - en particular, si x está indefinido, $x == x$ es indefinido (no es verdadero)

Especificación

Clase 3

Lenguaje de especificación

Funciones auxiliares

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

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

- f es el nombre de la función
- Puede usarse en el resto de la especificación en lugar de la expresión e
- Los parámetros son opcionales
- 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 $suc(x : \text{Int}) : \text{Int} = x + 1$;

Puedo usarla, por ejemplo, en la poscondición de un problema

Definir vs. especificar

- Definimos funciones auxiliares
  - Expresión del lenguaje a la que la función es equivalente
  - Esto permite usar la función dentro de las especificaciones
- 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 escribirla
    - Podría no haber ningún algoritmo que sirviera como solución
    - Si damos la solución va a ser en otro lenguaje (por ejemplo, de programación)
  - En la especificación de un problema o de un tipo no podemos usar otra función que hayamos especificado

Tipos enumerados

- Primer mecanismo para definir tipos propios
- Cantidad finita de elementos. Cada uno, representado 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)
- Opuesta: nombre u ord^{-1}

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;

Secuencias

- También se llaman listas
- Familia de tipos
  - Para cada tipo de datos hay un tipo secuencia
  - Tiene como elementos las secuencias de elementos de ese tipo
- Secuencia: varios elementos del mismo tipo, posiblemente repetidos, ubicados en un cierto orden
- 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

  secuencia de Int: $[1, 1+1, 3, 2*2, 3, 5]$
- La secuencia vacía (de elementos de cualquier tipo) se representa []
- Se puede formar secuencias de elementos de cualquier tipo
  - Como las secuencias de enteros son tipos, existen por ejemplo las secuencias de secuencias de enteros

    secuencia de secuencias de enteros:

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

Secuencias por comprensión

Elementos de otras secuencias que cumplan ciertas condiciones

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

- 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
- Resultado: Secuencia con el valor de la expresión calculado para todos los elementos seleccionados por los selectores que cumplen las condiciones

Ejemplo:

$[(x, y) | x \leftarrow [1, 2], y \leftarrow [1, 2, 3], x < y] == [(1, 2), (1, 3), (2, 3)]$

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 de la $\textit{expresión}_1$ y el de la $\textit{expresión}_2$ (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 uno de los extremos o ambos

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

Divisores comunes (asumir a y b positivos)

aux $divCom(a, b : \text{Int}) : [\text{Int}] =$

$$[x | x \leftarrow [1..a + b], divide(x, a), divide(x, b)];$$

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

Ejemplo:

$divCom(8, 12) == [1, 2, 4]$

Ejemplos de secuencias por comprensión

Cuadrados de los elementos impares

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

Suma de los elementos distintos

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: $long(a : [T]) : \text{Int}$
  - Longitud de la secuencia a
  - Notación: $long(a)$ se puede escribir $|a|$
- Indexación: $indice(a : [T], i : \text{Int}) : T$
  - requiere $0 \leq i < |a|$;
  - 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 $|a| > 0$;
  - Primer elemento de la secuencia

Más operaciones

- Cola: $cola(a : [T]) : [T]$
  - requiere $|a| > 0$;
  - 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: $cons(t : T, a : [T]) : [T]$
  - Una secuencia como a, agregándole t como primer elemento
  - Notación: $cons(t, a)$ se puede escribir $t : a$

Más operaciones

- Concatenación: $conc(a, b : [T]) : [T]$
  - Secuencia con los elementos de a, seguidos de los de b
  - Notación: $conc(a, b)$ se puede escribir $a + +b$
- Subsecuencia: $sub(a : [T], d, h : \text{Int}) : [T]$
  - 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 $0 \leq i < |a|$;
  - Igual a la secuencia a, pero el valor en la posición i es val

Subsecuencias con intervalos

- Notación para obtener una subsecuencia de una secuencia dada, 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 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 es Indef)

Operaciones 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 $\prod sec$

Para todo

$(\forall\ selectores,\ condiciones)\ expresión$

- Término de tipo Bool
- Afirma que todos los elementos de una lista por comprensión cumplen una propiedad
- Notación: en lugar de $\forall$ se puede escribir paratodo
- Equivale a $todos([expresión \mid selectores,\ condiciones])$
- Ejemplo: "todos los elementos en posiciones pares son mayores que 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

$(\exists\ selectores, condiciones)expresión$

- Hay algún elemento que cumple la propiedad
- Equivale a $alguno([expresión \mid selectores, condiciones])$
- Notación: en lugar de $\exists$ 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 querer contar cuántos elementos de una secuencia cumplen una condición
- Para eso, medimos la longitud de una secuencia definida por comprensión

Ejemplos:

- "¿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 usarla para saber si dos secuencias tienen los mismos elementos (en otro orden)

  aux $mismos(a, b : [T]) : \text{Bool} =$
  $(|a| == |b| \wedge (\forall c \leftarrow a)\ cuenta(c, a) == cuenta(c, b))$;
- "¿cuántos primos positivos hay que sean menores a n?"

  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

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

- Notación parecida a las secuencias por comprensión
- 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*
- *selectores* y *condición*: Como en las secuencias por comprensión
- *expresión*: También, pero puede (y suele) aparecer el acumulador
  - *acumulador* no puede aparecer 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* (de tipo *tipoAcum*)

Ejemplos de acumulación

- aux $sum(l : [\text{Float}]) : \text{Float} = acum(s+i \mid s : \text{Float} = 0, i \leftarrow l)$;
- Productoria
  aux $prod(l : [\text{Float}]) : \text{Float} =$
  $acum(p * i \mid p : \text{Float} = 1, i \leftarrow l)$;
- 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 $[1,1]$
  - por ejemplo, $fiboSuc(5) == [1,1,2,3,5,8]$
- n-ésimo número de Fibonacci ($n \geq 1$)
  aux $fibo(n : \text{Int}) : \text{Int} = (fiboSuc(n-1))[n-1]$;
  - por ejemplo, $fibo(6) == 8$

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, $concat([[1,2],[3],[4,5,6]]) == [1,2,3,4,5,6]$
- Secuencias por comprensión
  El término $[expresión \mid selectores, codición]$ es equivalente a
  $acum(res{+}{+}[expresión] \mid res : [T] = [], selectores, codición)$;

Especificación de problemas

- Problema: función a definir
- Especificación: contrato que debe cumplir la función para ser considerada solución del problema
- Hay que distinguir
  - el qué (especificación, el contrato a cumplir)
  - el cómo (implementación de la función)
- Sin referencias a posibles métodos para resolver el problema
- Distintas soluciones a un mismo problema
  - Distintos lenguajes de programación
  - Para el mismo lenguaje de programación, distintas formas de programar una función que cumpla con la especificación
  - Cada algoritmo posible es una solución distinta para el problema
- Conclusión: hay un conjunto (tal vez vacío) de algoritmos que cumplen cada especificación

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 una tupla
  - 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 $a == q * b + r \wedge 0 \leq r < b$;
  }

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 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 poscondición describe qué hacer y no cómo hacerlo
    No dice cómo calcular la raíz, ni qué raíz devolver
  - Sobrespecificación: poscondición que pone más restricciones de las necesarias (fija la forma de calcular la solución)
    Ejemplo: asegura $r == (-b + (b^2 - 4 * a * c)^{1/2})/(2 * a)$
    - esto sería sobreespecificar aun con $a \neq 0$ en la precondición

Otro ejemplo

Encontrar el índice (la posición) del menor elemento en una secuencia de números reales distintos no negativos

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: $(\forall i \leftarrow [0..|a|))\ a[i] \notin a[0..i)$;

Especificación

Clase 4

Tipos compuestos

Tipos compuestos

- cada valor de un tipo básico representa un elemento atómico, indivisible
  - Int
  - Float
  - Bool
  - Char
- un valor de un tipo compuesto contiene información que puede ser dividida en componentes de otros tipos
- ya vimos dos ejemplos de tipos compuestos:
  - secuencias: un valor de tipo secuencia de enteros tiene varios componentes, cada uno es un entero
  - tuplas: un valor de tipo par ordenado de enteros tiene dos componentes
- para definir un tipo compuesto, tenemos que darle
  - un nombre
  - uno o más observadores

Observadores

- funciones que se aplican a valores del tipo compuesto y devuelven el valor de sus componentes
  - pueden tener más parámetros
- definen el tipo compuesto
  - si dos términos del tipo dan el mismo resultado para todos los observadores, se los considera iguales
  - esta es la definición implícita de la igualdad $==$ para tipos compuestos (el $==$ está en todos los tipos)
- pueden tener precondiciones
  - si vale la precondición, no pueden devolver un valor indefinido
- no tienen poscondiciones propias
- si hay condiciones generales que deben cumplir los elementos del tipo, se las presenta como invariantes de tipo

Tipo Punto

- un punto en el plano
- componentes: coordenadas (x e y)
- un observador para obtener cada una

```
tipo Punto {
    observador X(p : Punto) : Float;
    observador Y(p : Punto) : Float
}
```

- especificación una función que recibe dos números reales y construye un punto con esas coordenadas:

```
problema nuevoPunto(a, b : Float) = res : Punto {
    asegura X(res) == a;
    asegura Y(res) == b;
}
```

Cuando el resultado es de un tipo compuesto, usamos los observadores para describir el resultado

- función auxiliar para calcular la distancia entre dos puntos

aux $dist(p, q : \text{Punto}) : \text{Float} = \left((X(p) - X(q))^2 + (Y(p) - Y(q))^2\right)^{1/2}$

Tipo Círculo

- sus componentes son de tipos distintos

```
tipo Círculo {
    observador Centro(c : Círculo) : Punto;
    observador Radio(c : Círculo) : Float;
    invariante Radio(c) > 0;
}
```

- especificar el problema de construir un círculo a partir de
  - su centro y su radio

```
problema nuevoCírculo(c : Punto, r : Float) = res : Círculo {
    requiere r > 0;
    asegura (Centro(res) == c ∧ Radio(res) == r);
}
```

  - su centro y un punto sobre la circunferencia:

```
problema nuevoCírculoPuntos(c, x : Punto) = res : Círculo {
    requiere dist(c, x) > 0;
    asegura (Centro(res) == c ∧ Radio(res) == dist(c, x));
}
```


Invariantes de tipo

- son la garantía de que los elementos estén bien construidos
- deben cumplirse para cualquier elemento del tipo
- los problemas que reciban valores del tipo compuesto como argumentos pueden suponer que se cumplen los invariantes

```
tipo T {
    observador ...;
        ⋮
    invariante R(x);
}
```

```
problema probA(..., x : T, ...) = ... {
    requiere ... ∧ R(x) ∧ ...;
                 (no hace falta)
    asegura ...;
}
```


Tipos genéricos

- en los ejemplos vistos, cada componente es de un tipo determinado de antemano
- también hay tipos compuestos que representan una estructura
  - su contenido van a ser valores no siempre del mismo tipo
  - comportamiento uniforme
- se llaman tipos genéricos o tipos paramétricos
  - definen una familia de tipos
    - cada elemento de la familia es una instancia del tipo genérico
  - los problemas que usen un tipo genérico definen una familia de problemas
- permiten definir funciones auxiliares o especificar problemas
  - genéricos: su descripción depende únicamente de la estructura del tipo genérico
  - específicos de un tipo: su descripción solo tiene sentido para un cierto tipo
- el nombre del tipo tiene parámetros
  - variables que representan a los tipos de los componentes
  - los parámetros de tipo se escriben entre los símbolos ⟨ y ⟩ después del nombre del tipo

Tipo Matriz$\langle T \rangle$

- tipo genérico de las matrices cuyos elementos pertenecen a un tipo cualquiera T

```
tipo Matriz⟨T⟩{
    observador filas(m : Matriz⟨T⟩) : Int;
    observador columnas(m : Matriz⟨T⟩) : Int;
    observador val(m : Matriz⟨T⟩, f, c : Int) : T {
        requiere 0 ≤ f < filas(m);
        requiere 0 ≤ c < columnas(m);
    }                                   } precondición del observador
    invariante filas(m) > 0;
    invariante columnas(m) > 0;
}
```

- ejemplo: una instancia del tipo genérico Matriz$\langle T \rangle$ es Matriz$\langle$Char$\rangle$
  - el tipo de matrices cuyos elementos son caracteres
- un tipo genérico define una familia (infinita) de tipos
  - algunos miembros de la familia Matriz$\langle T \rangle$ son

    Matriz$\langle$Int$\rangle$, Matriz$\langle$Punto$\rangle$, Matriz$\langle$Matriz$\langle$Float$\rangle\rangle$

Operaciones genéricas con matrices

- expresión que cuenta la cantidad de elementos de una matriz

```
aux elementos(m : Matriz⟨T⟩) : Int = filas(m) * columnas(m);
```

- especificación del problema de cambiar el valor de una posición de una matriz

```
problema cambiar(m : Matriz⟨T⟩, f, c : Int, v : T) {
    requiere 0 ≤ f < filas(m);
    requiere 0 ≤ c < columnas(m);
    modifica m;
    asegura filas(m) == filas(pre(m));
    asegura columnas(m) == columnas(pre(m));
    asegura val(m, f, c) == v;
    asegura (∀i ← [0..filas(m)))
                (∀j ← [0..columnas(m)), ¬(i == f ∧ j == c))
                val(m, i, j) == val(pre(m), i, j);
}
```


Operaciones sobre matrices para tipos instanciados

- expresión que suma los elementos de una matriz de enteros

```
aux suma(m : Matriz⟨Int⟩) : Int =
    ∑[val(m, i, j) | i ← [0..filas(m)), j ← [0..columnas(m))];
```

- especificación del problema de construir la matriz identidad de $n \times n$:

```
problema matId(n : Int) = ident : Matriz⟨Int⟩ {
    requiere n > 0;
    asegura filas(ident) == columnas(ident) == n;
    asegura (∀i ← [0..n)) val(ident, i, i) == 1;
    asegura (∀i ← [0..n))(∀j ← [0..n), i ≠ j) val(ident, i, j) == 0;
}
```

  - ¿importa el orden?
  - sí; si estuviera al revés, se podría indefinir $val(ident, i, i)$
  - en este caso, se indefiniría la poscondición, valiendo la precondición

El tipo Secuencia$\langle T \rangle$

- ya lo usamos con su nombre alternativo: $[T]$
- presentamos también sus observadores: longitud e indexación

```
tipo Secuencia⟨T⟩{
    observador long(s : Secuencia⟨T⟩) : Int;
    observador índice(s : Secuencia⟨T⟩, i : Int) : T{
        requiere 0 ≤ i < long(s); ⟶ precondición de observador
    }
}
```

- notaciones alternativas
  - $|s|$ para la longitud
  - s_i o $s[i]$ para la indexación

Operaciones genéricas con secuencias

- podemos definir
  - aux $ssc(a, b : [T]) : Bool =$
    $(\forall i \leftarrow [0..|b| - |a|))\ a == b[i..(i + |a|))$;
  - aux $cab(a : [T]) : T = a[0]$;
  - aux $cola(a : [T]) : [T] = a[1..|a|)$;
  - aux $en(t : T, a : [T]) : \text{Bool} = [x \mid x \leftarrow a, x == t] \neq [\,]$;
  - aux $sub(a : [T], d, h : \text{Int}) : [T] =$
    $[a[i] \mid i \leftarrow [d..h], (0 \geq \wedge h < |a|)]$;
  - aux $todos(sec : [\text{Bool}]) : \text{Bool} = \text{false} \notin sec$;
  - aux $alguno(sec : [\text{Bool}]) : \text{Bool} = \text{true} \in sec$;
- para algunas no usamos directamente los observadores
- aprovechamos la notación de listas por comprensión
- el selector es parte de la notación de listas por comprensión (estructura especial de nuestro lenguaje de especificación)

Operaciones sobre secuencias para tipos instanciados

- para representar textos (cadenas de caracteres) usamos el tipo Secuencia⟨Char⟩ (o [Char])
- queremos ver si alguna palabra de una lista aparece en un libro

  problema $hayAlguna(palabras : [[\text{Char}]], libro : [\text{Char}]) = res : \text{Bool}\{$
  requiere $NoVacías : (\forall p \leftarrow palabras)\ |p| > 0$;
  requiere $SinEspacios : \neg(\exists p \leftarrow palabras)\ ' \ ' \in p$;
  asegura $res == (\exists p \leftarrow palabras)\ ssc(p, libro)$;
  $\}$
- lista de palabras: tipo [[Char]] o Secuencia⟨Secuencia⟨Char⟩⟩
  - secuencias que en cada posición tienen una secuencia de caracteres
- precondiciones
  - cada palabra debe tener al menos una letra
  - no contienen ningún espacio (dejarían de ser palabras)
- en vez de SinEspacios: que las palabras no contengan ningún símbolo que no sea una letra

  requiere SinSímbolos: $(\forall p \leftarrow palabras)\ soloLetras(p)$;
  aux letras: $[Char] = [\text{'}A\text{'}..\text{'}Z\text{'}] + +[\text{'}a\text{'}..\text{'}z\text{'}]$;
  aux soloLetras($s : [\text{Char}]$) : Bool $= (\forall l \leftarrow s)\ l \in letras$;

Tuplas

- ya las mencionamos
- secuencias de tamaño fijo
- cada elemento puede pertenecer a un tipo distinto
- ejemplos: pares, ternas

  tipo Par$\langle A, B\rangle\{$
  observador prm($p : Par\langle A, B\rangle$) : A;
  observador sgd($p : Par\langle A, B\rangle$) : B;
  $\}$

  tipo Terna$\langle A, B, C\rangle\{$
  observador prm3(t : Terna$\langle A, B, C\rangle$) : A;
  observador sgd3(t : Terna$\langle A, B, C\rangle$) : B;
  observador trc3(t : Terna$\langle A, B, C\rangle$) : C;
  $\}$
- notación
  - Par$\langle A, B\rangle$ también se puede escribir (A, B)
  - Terna$\langle A, B, C\rangle$ también se puede escribir (A, B, C)

ifThenElse⟨T⟩

Función que elige entre dos elementos del mismo tipo, según una condición (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 : Int) : Int = ifThenElse⟨Int⟩$(a > b, a, b)$

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

  aux máx(a, b : Int) : Int = ifThenElse$(a > b, a, b)$ o bien
  aux máx(a, b : Int) : Int = if $a > b$ then a else b
- expresión que dado x devuelve $1/x$ si $x \neq 0$ y 0 sino

  aux unoSobre(x : Float) : Float = $\underbrace{\text{if } x \neq 0 \text{ then } 1/x \text{ else } 0}_{\text{no se indefine cuando } x = 0}$

Más aplicaciones de IfThenElse

- agregar un elemento como primer elemento de una lista

  aux cons($x : T, a : [T]) : [T] =$
  $[\text{if } i == -1 \text{ then } x \text{ else } a[i] \mid i \leftarrow [-1..|a|)];$

- concatenar dos listas

  aux conc$(a, b : [T]) : [T] =$
  $[\text{if } i < |a| \text{ then } a[i] \text{ else } b[i - |a|] \mid i \leftarrow [0..|a| + |b|)];$

- cambiar el valor de una posición

  aux cambiar$(a : [T], i : \text{Int}, v : T) : [T] =$
  $[\text{if } i \neq j \text{ then } a[i] \text{ else } v \mid j \leftarrow [0..|a|)];$

Programación funcional

Clase 1

Funciones Simples - recursión - tipos de datos

Algoritmos y programas

- aprendieron a especificar problemas
- el objetivo es ahora escribir un algoritmo que cumpla esa especificación
  - secuencia de pasos que pueden llevarse a cabo mecánicamente
- puede haber varios algoritmos que cumplan una misma especificación
- una vez que se tiene el algoritmo, se escribe el programa
  - expresión formal de un algoritmo
  - lenguajes de programación
    - sintaxis definida
    - semántica definida
    - qué hace una computadora cuando recibe ese programa
    - qué especificaciones cumple
    - ejemplos: Haskell, C, C++, C#, Java, Smalltalk, Prolog, etc.

Paradigmas

- paradigmas de programación
  - formas de pensar un algoritmo que cumpla una especificación
  - cada uno tiene asociado un conjunto de lenguajes
  - nos llevan a encarar la programación según ese paradigma
- Haskell: paradigma de programación funcional
  - programa = colección de funciones
    - aparatos que transforman datos de entrada en un resultado
  - los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo calcular esas funciones

Expresiones

- tira de símbolos que representan (denotan) un valor
- ejemplos:
  - 2
  - `1+1`
  - `(3*7+1)/11`
  - todas representan el mismo valor
- los valores se agrupan en tipos
  - como en el lenguaje de especificación: `Int`, `Float`, `Bool`, `Char`
  - hay también tipos compuestos (por ejemplo, pares ordenados)

Transparencia referencial

- propiedad muy importante de la programación funcional

  Una expresión representa siempre el mismo valor en cualquier lugar de un programa

  - otros paradigmas: el significado de una expresión depende del contexto
- muy útil al modificar programas
  - modificar una parte no afecta otras
- también para verificar (demostrar que se cumple la especificación)
  - podemos usar propiedades ya probadas para sub expresiones
  - valor no depende de la historia
  - valen en cualquier contexto

Formación de expresiones

- expresiones atómicas (las más simples)
  - también se llaman formas normales
  - son la forma más intuitiva de representar un valor
  - ejemplos
    - `2`
    - `False`
    - `(3, True)`
  - es común llamarlas "valores"
    - aunque no son un valor, *representan* un valor, como las demás expresiones
- expresiones compuestas
  - se construyen combinando expresiones atómicas con operaciones
  - ejemplos:
    - `1+1`
    - `1==2`
    - `(4-1, True || False)`

Expresiones mal formadas

- algunas cadenas de símbolos no forman expresiones
  - por problemas sintácticos
    - `+*1-`
    - `(True`
    - `('a',)`
  - o por error de tipos
    - `2 + False`
    - `2 || 'a'`
    - `4 * 'b'`
- para saber si una expresión está bien formada, aplicamos
  - reglas sintácticas
  - reglas de asignación de tipos (o de inferencia de tipos)

Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores)

- una función es un valor
- la operación básica que podemos realizar con ese valor es la aplicación
  - aplicar la función a un elemento para obtener un resultado
- sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro)
  - `f` es una función y `e` un elemento de su conjunto de partida
  - `f e` denota el elemento que se relaciona con `e` por medio de la función `f`
- por ejemplo, `doble 2` representa al número 4

Ecuaciones

- Dada una expresión, ¿cómo sabemos qué valor denota?
- Usando las ecuaciones que la definen
  - por ejemplo: `doble x = x + x`
- se reemplaza cada sub expresión por otras según las ecuaciones
- pero este proceso puede no terminar
  - aún con ecuaciones bien pensadas
    `doble (1 + 1)`
  - reemplazo `1 + 1` por `doble 1`
    `doble (doble 1)`
  - reemplazo `doble 1` por `1 + 1`
  - volví a empezar...

Ecuaciones orientadas

- solución: usar ecuaciones orientadas
  - lado izquierdo: expresión a definir
  - lado derecho: definición
- cálculo de valores = reemplazar subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Ejemplo: `doble x = x + x`

`doble (1 + 1)` $\rightsquigarrow$ `(1 + 1) + (1 + 1)` $\rightsquigarrow$ `2 + (1 + 1)` $\rightsquigarrow$ `2 + 2` $\rightsquigarrow$ `4`

También podría ser:

`doble (1 + 1)` $\rightsquigarrow$ `doble 2` $\rightsquigarrow$ `2 + 2` $\rightsquigarrow$ `4`

Más adelante veremos cómo funciona Haskell en particular.

- reducción = reemplazar una subexpresión por su definición, sin tocar el resto
  - la expresión resultante puede no ser más corta
  - pero seguramente está "más definida"
    - más cerca de ser una forma normal

Programa funcional

- conjunto de ecuaciones que definen una o más funciones
- ¿para qué se usa un programa funcional?
  - para reducir expresiones
  - puede no ser tan claro que esto resuelva un problema
  - las ecuaciones orientadas, junto con el mecanismo de reducción describen algoritmos: pasos para resolver un problema

Ejemplos:

- `doble x = x+x`
- `fst (x,y) = x`
- `dist (x,y) = sqrt (x^2+y^2)`
- ```
  signo 0 = 0
  signo x | x > 0 = 1
  signo x | x < 0 = -1
  ```
- `promedio1 (x,y) = (x+y)/2`
- `promedio2 x y = (x+y)/2`
- ```
  fact 0 = 1
  fact n | n > 0 = n * fact (n-1)
  ```
- ```
  fib 1 = 1
  fib 2 = 1
  fib n | n > 2 = fib (n-1) + fib (n-2)
  ```
  ```

Definiciones recursivas

- en el cuerpo de la definición de `fact` y `fib` (lado derecho) aparece el nombre de la función
- propiedades de la definición
  - tiene que tener uno o más casos bases
    - en el caso de `fact` el caso base es

      `fact 0 = 1`
    - en el caso de `fib` los casos bases son

      `fib 1 = 1` y `fib 2 = 1`
  - las llamadas recursivas del lado derecho tienen que acercarse más al caso base
    - en el caso de `fact` la llamada recursiva es

      `fact (n-1)`
    - en el caso de `fib` las llamadas recursivas son

      `fib (n-1)` y `fib (n-2)`

Asegurarse de llegar a un caso base

Supongamos esta especificación:

problema $par(n : \text{Int}) = result : \text{Bool}\{$
  requiere $n \geq 0$;
  asegura $result == (n \text{ mód } 2 == 0)$;
$\}$

- ¿este programa cumple con la especificación?

```
par 0 = True
par n = par (n-2)
```

  - no, porque se indefine para los impares positivos
- se arregla de alguna de estas formas:

```
par 0 = True
par 1 = False
par n = par (n-2)
```

```
par 0 = True
par n = not (par (n-1))
```


Correctitud

- para demostrar que una definición es correcta, hace falta una especificación y un programa:

problema $fact(n : \text{Int}) = result : \text{Int}\{$
  requiere $n \geq 0$;
  asegura $result == \prod[1..n]$;
$\}$

```
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

- la especificación (o cualquier otro comentario) se puede escribir en el mismo programa funcional precediendo cada linea con -- (doble menos)
  - no se considera parte del programa
  - por ejemplo:

```
-- problema fact (n: Int) = result: Int {
-- requiere n >= 0;
-- asegura result == prod [1..n];
-- }
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```


Demostración de correctitud

Si la definición es recursiva, lo más natural es demostrar por inducción

- caso base ($n == 0$)
  - la especificación dice que hay que probar que la función `fact` con entrada 0 devuelve $\prod[1..0] == \prod[\,] = 1$
  - pero eso es justamente lo que devuelve `fact` en el caso base
- hipótesis inductiva (HI): $\texttt{fact}(n) == \prod[1..n]$
  - la especificación dice que hay que probar que la función `fact` con entrada $n+1$ devuelve $\prod[1..n+1]$
  - calculemos $\texttt{fact}(n+1)$
  - gracias a la precondición, $n+1 > 0$. Entonces uso la segunda ecuación instanciando en $n+1$

$$fact(n+1) == (n+1) * fact(n)$$

  - por HI:

$$\begin{aligned} fact(n+1) &== (n+1) * \prod[1..n] \\ &== \prod[1..(n+1)] \end{aligned}$$

  - entonces, demostré que `fact` con entrada $n+1$ devuelve $\prod[1..(n+1)]$

Tipos de datos

- es un concepto que vimos en el lenguaje de especificación
- conjunto de valores a los que les pueden aplicar las mismas operaciones
- en Haskell, todo valor pertenece a algún tipo
  - las funciones son valores, y también tienen tipo
  - ejemplo: el tipo "funciones de enteros en enteros"
- todo valor pertenece a un tipo. Toda expresión (bien formada) denota un valor. Entonces, toda expresión tiene un tipo (el del valor que representa)

Haskell es un lenguaje fuertemente tipado

- no se pueden pasar elementos de un tipo a una operación que espera argumentos de otro

También tiene tipado estático

- no hace falta hacer funcionar un programa para saber de qué tipo son sus expresiones
- el intérprete puede controlar si un programa tiene errores de tipos

Notación para tipos

- `e :: T`
  - la expresión `e` es de tipo `T`
  - el valor denotado por `e` pertenece al conjunto de valores llamado `T`
- ejemplos:
  - `2 :: Int`
  - `False :: Bool`
  - `'b':: Char`
  - `doble :: Int -> Int`
- sirve para escribir reglas y razonar sobre Haskell
- también se usa dentro de Haskell
  - indica de qué tipo queremos que sean los nombres que definimos
  - el intérprete chequea que el tipo coincida con el de las expresiones que lo definen
  - podemos obviar las declaraciones de tipos pero nos perdemos la oportunidad del doble chequeo
  - existen casos en los que sí es obligatorio, para dirimir ambigüedades

Polimorfismo

- el tipado de Haskell es fuerte
- pero hay funciones que pueden aplicarse a distintos tipos de datos (sin redefinirlas)
- se llama polimorfismo
  - se usa cuando el comportamiento de la función no depende del valor de sus parámetros
  - lo vimos en el lenguaje de especificación con las funciones genéricas
- ejemplo: la función identidad: `id x = x`
  - ¿de qué tipo es `id`?
    - `id 3`: Por las reglas, `id :: Int -> Int`
    - `id True`: Por las reglas, `id :: Bool -> Bool`
    - `id doble`: Por las reglas, `id :: (Int -> Int) -> (Int -> Int)`
  - la idea es `id :: a -> a`
    - sin importar qué tipo sea a
    - en Haskell se escribe usando variables de tipo.
    - "id es una función que dado un elemento de algún tipo a devuelve otro elemento de ese mismo tipo"
    - indica que `id` es de de un tipo paramétrico (depende de un parámetro)
    - `id` es una función polimórfica

Declaraciones de tipo

- `doble :: Int -> Int`
  ```
  doble x = x+x
  ```
- `fst :: (a,b) -> a`
  ```
  fst (x,y) = x
  ```
- `dist :: (Float, Float) -> Float`
  ```
  dist (x,y) = sqrt (x^2+y^2)
  ```
- `signo Int -> Int`
  ```
  signo 0 = 0
  signo x | x > 0 = 1
  signo x | x < 0 = -1
  ```
- `promedio1 :: (Float, Float) -> Float`
  ```
  promedio1 (x,y) = (x+y)/2
  ```
- `promedio2 :: Float -> Float -> Float`
  ```
  promedio2 x y = (x+y)/2
  ```
- `fact :: Int -> Int`
  ```
  fact 0 = 1
  fact n | n > 0 = n * fact (n-1)
  ```
- `fact :: Int -> Int`
  ```
  fib 1 = 1
  fib 2 = 1
  fib n | n > 2 = fib (n-1) + fib (n-2)
  ```

Currificación

- diferencia entre `promedio1` y `promedio2`
  - ```
    promedio1 :: (Float, Float) -> Float
    promedio1 (x,y) = (x+y)/2
    ```
  - ```
    promedio2 :: Float -> Float -> Float
    promedio2 x y = (x+y)/2
    ```
- solo cambia el tipo de datos de la función
  - promedio1 recibe un solo parámetro (un par)
  - promedio2 recibe dos Float separados por un espacio
  - para declararla, separamos los tipos de los parámetros con una flecha
  - tiene motivos teóricos y prácticos (que no veremos ahora)
- la notación se llama currificación en honor al matemático Haskell B. Curry
- para nosotros, alcanza con ver que evita el uso de varios signos de puntuación (comas y paréntesis)
  - `promedio1 (promedio1 (2, 3), promedio1 (1, 2))`
  - `promedio2 (promedio2 2 3) (promedio2 1 2)`

Programación funcional

Clase 2

Tipos algebraicos

Tipos algebraicos y abstractos

- ya vimos los tipos básicos
  - `Int`
  - `Float`
  - `Char`
  - `Bool`
- otros tipos de datos:
  - tipos algebraico
  - tipos abstracto
- tipo algebraico
  - conocemos la forma que tiene cada elemento
  - tenemos un mecanismo para inspeccionar cómo está construido cada dato
- tipo abstracto
  - solo conocemos sus operaciones
  - no sabemos cómo están formados los elementos
  - la única manera de obtener información sobre ellos es mediante las operaciones

Tipos algebraicos

- para crear un tipo algebraico decimos qué forma va a tener cada elemento
- se hace definiendo constantes que se llaman constructores
  - empiezan con mayúscula (como los tipos)
  - pueden tener argumentos, pero no hay que confundirlos con funciones
    - no tienen reglas de inferencia asociada
    - forman expresiones atómicas (valores)
- ejemplo muy sencillo de tipo algebraico: Bool
  - tiene dos constructores (sin argumentos)

```
True :: Bool
False :: Bool
```

Definición de tipos algebraicos

- cláusulas de definición de tipos algebraicos
  - empiezan con la palabra `data`
  - definen el tipo y sus constructores
  - cada constructor da una alternativa distinta para construir un elemento del tipo
  - los constructores se separan entre sí por barras verticales
- `data Sensación = Frío | Calor`
  - tiene dos constructores sin parámetros
  - el tipo tiene únicamente dos elementos, como el tipo Bool
- `data Figura = Círc Float | Rect Float Float`
  - dos constructores con parámetros
  - algunas figuras son círculos y otras rectángulos
    - los círculos se diferencian por un número (su radio)
    - los rectángulos, por dos (su base y su altura)
  - ejemplos:
    - `c1 = Círc 1`
    - `c2 = Círc (4.5 - 3.5)`
    - `círculo x = Círc (x+1)`
    - `r1 = Rect 2.5 3`
    - `cuadrado x = Rect x x`

Pattern matching

- correspondencia o coincidencia de patrones
- mecanismo para ver cómo está construido un elemento de un tipo algebraico
- si definimos una función que recibe como parámetro una `Figura`, podemos averiguar si es un círculo o un rectángulo (y con qué parámetros fue construida)
- patterns: expresiones del lenguaje formadas solamente por constructores y variables que no se repiten
  - `Rect x y` es un patrón
  - `3 + x` no es un patrón
  - `Rect x x` tampoco porque tiene una variable repetida
- matching: operación asociada a un patrón
  - dada una expresión cualquiera dice si su valor coincide por su forma con el patrón
  - si la correspondencia existe, entonces liga las variables del patrón a las subexpresiones correspondientes

Ejemplo

```
área :: Figura -> Float

área (Círc radio)       = pi * radio * radio
área (Rect base altura) = base * altura

círculo :: Float -> Figura
círculo x = Círc (x+1)
```

- lado izquierdo: función que estamos definiendo aplicada a un patrón
- evaluemos la expresión `área (círculo 2)`
  - el intérprete debe elegir cuál de las ecuaciones de `área` utilizar
  - primero debe evaluar `círculo 2` para saber a qué constructor corresponde
  - la reducción da `Círc (2+1)`
  - ya se puede verificar cada ecuación de `área` para buscar el matching
  - se logra con la primera ecuación
    - `radio` queda ligada a `(2+1)`
- luego de varias reducciones (aritméticas) más, se llega al valor de la expresión: 28.2743

Sinónimos de tipos

- se usa la cláusula `type` para darle un nombre nuevo a un tipo existente
  - no se crea un nuevo tipo, sino un sinónimo de tipo
  - los dos nombres son equivalentes
- ejemplo: nombrar una instancia particular de un tipo paramétrico: `type String = [Char]`
- ejemplo: renombrar un tipo existente con un nombre más significativo:

```
type Nombre = String
type Sueldo = Int
type Empleado = (Nombre, Sueldo)
type Dirección = String
type Persona = (Nombre, Dirección)
```

- `Persona` es un par de `String`, pero `(String, String)`, es más difícil de entender
- también hay sinónimos de tipos paramétricos:

```
type IntY a = (Int, a)
```

Tipos recursivos

El tipo definido es argumento de alguno de los constructores

Ejemplo:

- `data N = Z | S N`
- tiene dos constructores:
  1. `Z` es un constructor sin argumentos
  2. `S` es un constructor con argumentos (de tipo `N`)
- elementos del tipo `N`:

```
Z   ,  S Z  ,  S (S Z)  ,  S (S (S Z))  ,  ...
↓      ↓       ↓           ↓
0      1       2           3
```

Este tipo puede representar a los números naturales.

Ejemplos

Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante recursión estructural.

- `suma :: N -> N -> N`

```
suma n Z     = n
suma n (S m) = S (suma n m)
```

- `producto :: N -> N -> N`

```
producto n Z     = Z
producto n (S m) = suma n (producto n m)
```

- `menorOIgual :: N -> N -> Bool`

```
menorOIgual Z     m     = True
menorOIgual (S n) Z     = False
menorOIgual (S n) (S m) = menorOIgual n m
```


Otro ejemplo

- `data P = T | F | A P P | N P`
- tiene cuatro constructores:
  1. `T` y `F` son constructores sin argumentos
  2. `A` es un constructor con dos argumentos (de tipo `P`)
  3. `N` es un constructor con un argumento (de tipo `P`)
- elementos del tipo `P`:

```
T     ,   F     ,    A T F      ,   N (A T F)      ,  ...
↓         ↓          ↓               ↓
true      false      (true ∧ false)  ¬(true ∧ false)
```

Este tipo puede representar a las fórmulas proposicionales.

Ejemplos

- `contarAes :: P -> Int`

```
contarAes T       = 0
contarAes F       = 0
contarAes (N x)   = contarAes x
contarAes (A x y) = 1 + (contarAes x) + (contarAes y)
```

- `valor :: P -> Bool`

```
valor T       = True
valor F       = False
valor (N x)   = not (valor x)
valor (A x y) = (valor x) && (valor y)
```

Listas

- se usan mucho (el igual que en el lenguaje de especificación)
- son un tipo algebraico recursivo paramétrico
  - en especificación las vimos definidas con observadores
  - en Haskell, se definen con constructores
- primero, vamos a definirlas como un tipo algebraico común

```
data List a = Nil | Cons a (List a)
```

- interpretamos
  - `Nil` como la lista vacía
  - `Cons x l` como la lista que resulta de agregar `x` como primer elemento de `l`
- por ejemplo,
  - `List Int` es el tipo de las listas de enteros. Son de este tipo:
    - `Nil`
    - `Cons 2 Nil`
    - `Cons 3 (Cons 2 Nil)`
  - `List (List Int)` es el tipo de las listas de listas de enteros. Son de este tipo:
    - `Nil`
    - `Cons Nil Nil`
    - `Cons (Cons 2 Nil) (Cons Nil Nil)`

Ejemplos

- calcular la longitud de una lista

```
longitud :: List a -> Int
longitud Nil         = 0
longitud (Cons x xs) = 1 + (longitud xs)
```

- sumar los elementos de una lista de enteros

```
sumar :: List Int -> Int
sumar Nil         = 0
sumar (Cons x xs) = x + (sumar xs)
```

- concatenar dos listas

```
concat :: List a -> List a -> List a
concat Nil         ys = ys
concat (Cons x xs) ys = Cons x (concat xs ys)
```


Notación de listas en Haskell

- `List a` se escribe `[a]`
- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`
- los constructores son `:` y `[]`
  - `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil)))`
  - `(2 : (3 : (2 : (0 : []))))`
  - `2 : 3 : 2 : 0 : []`
- notación más cómoda: `[2,3,2,0]`

Ejemplos (todas están en el preludio)

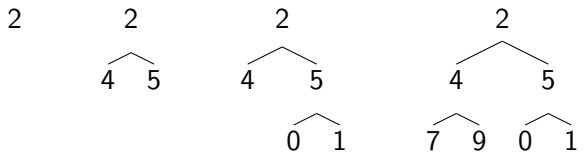
-
```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```
-
```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```
-
```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

`++` concatena dos listas. Es un operador que se usa con notación infija $[1,2]++[3,4,5]$

Árboles

- estructuras formadas por nodos
- almacenan valores de manera jerárquica (cada nodo guarda un valor)
- vamos a trabajar con árboles binarios
  - de cada nodo salen cero o dos ramas
  - hoja: es un nodo que no tiene ramas

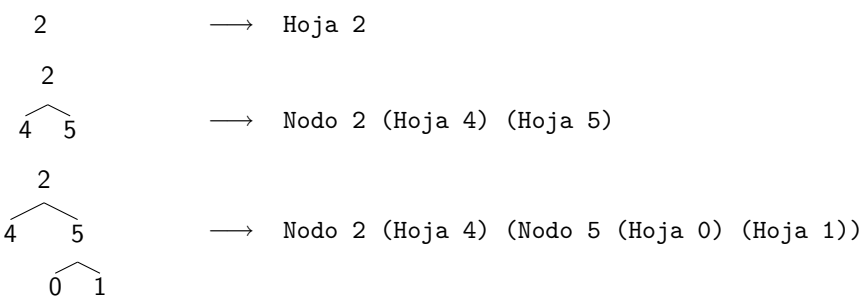
Ejemplos:



Definición del tipo Árbol

Definición del tipo

```
data Árbol a = Hoja a | Nodo a (Árbol a) (Árbol a)
```




```
hojas :: Árbol a -> Int
hojas (Hoja x)     = 1
hojas (Nodo x i d) = (hojas i) + (hojas d)
altura :: Árbol a -> Int
altura (Hoja x)     = 1
altura (Nodo x i d) = 1 + ((altura i) 'max' (altura d))
```


Recorridos en árboles

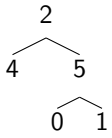
- las funciones que vimos van recorriendo un árbol y operando con cada nodo
- podrían hacerlo en cualquier orden con el mismo resultado
- a veces es importante el orden al recorrer un árbol para aplicarle alguna operación

```
inOrder, preOrder :: Árbol a -> [a]

inOrder (Hoja x)     = [x]
inOrder (Nodo x i d) = (inOrder i) ++ [x] ++ (inOrder d)

preOrder (Hoja x)     = [x]
preOrder (Nodo x i d) = x : ((preOrder i) ++ (preOrder d))
```

Por ejemplo, para el árbol `A`




```
inOrder A     es     [4,2,0,5,1]
preOrder A    es     [2,4,5,0,1]
```


Programación funcional

Clase 3

Tipos abstractos

Tipos abstractos

- Vimos cómo crear y cómo usar tipos de datos algebraicos
- También se puede recibir un tipo como abstracto
  - No se puede acceder a su representación
- No hace falta más de un programador
  - Creo un tipo (algebraico) y oculto su representación
  - En otras partes del programa me está prohibido accederla
- En el diseño de lenguajes de programación es un punto clave
  - Facilitar que el programador haga lo que quiere, pero alentarlo a hacerlo bien
  - Y, más importante, impedirle hacerlo mal: usar la representación interna del tipo en otros programas
- Dos motivos
  - Criterios básicos para tipos algebraicos
  - Mantenimiento

Criterios para tipos algebraicos

1. Toda expresión del tipo representa un valor válido
   - Constructor
   - Valores cualesquiera para sus parámetros
2. Igualdad por construcción
   - Dos valores son iguales solamente si se construyen igual
     - Mismo constructor
     - Mismos argumentos

Condiciones ideales

- A veces se construyen tipos algebraicos sin respetarlas
  - `Círc (-1.2)`
  - `Rect 2.3 4.5` $\neq$ `Rect 4.5 2.3`

Ejemplo de tipo algebraico: Complejo

- Toda combinación de dos Float es un complejo
- Dos complejos son iguales sii sus partes reales y sus partes imaginarias coinciden

```
data Complejo = C Float Float

parteReal, ParteImag :: Complejo -> Float

ParteReal (C r i) = r
ParteImag (C r i) = i

hacerPolar :: Float -> Float -> Complejo
hacerPolar rho theta =
        C (rho * cos theta) (rho * sin theta)
```


Racionales

```
data Racional = R Int Int

numerador, denominador :: Racional -> Int

numerador (R n d)  = n

denominador (R n d) = d
```

¿Está bien definido? ¡No!

- No todo par de enteros es un racional: R 1 0
- Hay racionales iguales con distinto numerador y denominador: R 4 2 y R 2 1

Racionales como tipo algebraico

Se puede usar, pero con mucho cuidado

- Al construir
  - Nunca segunda componente 0
- Al definir funciones
  - Mismo resultado para toda representación del mismo racional
  - Función `numerador` (devuelve la primera componente)
    - Resultados distintos para `R (-1) (-2)` y `R 2 4`
    - Son el mismo número, no estamos representando las fracciones

Tipos abstractos

- Ayuda del lenguaje
- La representación interna se usa en pocas funciones
- Mensaje de error si se intenta violar ese acuerdo

Mantenimiento

- Acceso por pattern matching limitado a pocas funciones
  - Si cambia la representación interna, solamente hay que cambiar esas funciones
  - En lo posible, muy cerca una de otra en el código (mismo archivo)
- Si no pedimos al lenguaje esta protección
  - Riesgo de usar la implementación en otros programas
  - Cuando la cambiemos, dejan de funcionar
  - Hay que modificar uno por uno

Uso de tipos abstractos

- Se recibe un tipo de datos abstracto
  - nombre del tipo
  - nombres de sus operaciones básicas
  - tipos de las operaciones
  - especificación de su funcionamiento
    - puede ser más o menos formal
    - en Haskell no es chequeada por el lenguaje
- ¿Cómo se utiliza el tipo?
  - a través de sus operaciones y únicamente así
  - no se puede usar pattern matching

Racionales como tipo abstracto

- Recibimos operaciones, que no sabemos cómo están implementadas

```
crearR :: Int -> Int -> Racional
numerR :: Racional -> Int
denomR :: Racional -> Int
```

- Y especificación

```
tipo Racional {
    observador numerR(r : Racional): Int;
    observador denomR(r: Racional): Int;
    invariante denomR(r) > 0;
    invariante mcd(numerR(r), denom(R)) == 1;
}
problema crearR(n, d: Int) = rac: Racional {
    requiere d ≠ 0;
    asegura numerR(rac) * d == denomR(rac) * n;
}
```


Racionales como tipo abstracto

- Numerador y denominador: forma normalizada (máxima simplificación) con signo en el numerador
- No sabemos cuándo se produce la simplificación para normalizar
  - al construir el número en `crearR`
  - al evaluarlo, en `numerR` y `denomR`

Definición de nuevas operaciones

- Tal vez incluya operaciones básicas
  - Suma, multiplicación, división
- También podemos definirlas nosotros:

```
sumaR, multR, divR :: Racional -> Racional -> Racional

r1 `sumaR` r2 = crearR
    (denomR r2 * numerR r1 + denomR r1 * numerR r2)
    (denomR r1 * denomR r2)

r1 `multR` r2 = crearR
    (numerR r1 * numerR r2)
    (denomR r1 * denomR r2)

r1 `divR` r2 = crearR
    (denomR r2 * numerR r1)
    (denomR r1 * numerR r2)
```


Otro ejemplo: Conjuntos

- Teoría de conjuntos
  - Herramienta muy poderosa para el razonamiento matemático
- Pero el tipo de datos preferido para representar colecciones no es conjuntos, sino listas
- Motivo
  - Los conjuntos no pueden representarse con un tipo algebraico que cumpla los criterios
- Solución
  - Crear un tipo abstracto para los conjuntos
  - Las operaciones disponibles para el usuario van a limitar su uso

Operaciones de conjuntos

Vamos a definir conjuntos de enteros

- El conjunto vacío
  `vacío :: IntSet`
- ¿El conjunto dado es vacío?
  `esVacío :: IntSet -> Bool`
- ¿Un elemento pertenece al conjunto?
  `pertenece :: Int -> IntSet -> Bool`
- Agregar un elemento al conjunto, si no estaba. Si estaba, dejarlo igual
  `agregar :: Int -> IntSet -> IntSet`
- Elegir el menor número y quitarlo del conjunto
  `elegir :: IntSet -> (Int, IntSet)`

Transparencia referencial

- Las descripciones usan la metáfora de la modificación
  - Las operaciones no modifican los conjuntos, construyen conjuntos nuevos
  - Pero es fácil explicar las funciones en estos términos
- `elegir` no quita nada
  - El original, sigue igual
  - Devuelve un elemento y otro conjunto, con todos los elementos del primero menos ese
  - No queda claro qué pasa si el conjunto es vacío
  - Se puede suponer que da error
- Estas descripciones no sustituyen a una especificación

Nuevas operaciones

- Definamos una operación nueva: `unión`

```
unión :: IntSet -> IntSet -> IntSet

unión p q | esVacío p = q
unión p q | otherwise = uniónAux (elegir p) q

uniónAux (x, p') q = agregar x (unión p'q)
```

- Pudimos hacerlo sin conocer la representación de los conjuntos
  - Usamos las operaciones provistas
- Si cambia la representación
  - Habrá que rescribir las ecuaciones para las operaciones del tipo abstracto (`vacío`, `esVacío`, `pertenece`, `agregar`, `elegir`)
  - `unión` y cualquier otra definida en otros programas quedan intactas
- La recursión no es privativa del pattern matching
  - `unión` está definida en forma recursiva (a través de `uniónAux`)
  - Pero no usa recursión estructural ¡ni siquiera conocemos la estructura!

Creación de tipos abstractos

- Escribir un tipo abstracto que puedan usar otros programadores
- En Haskell se hace con módulos
  - Archivos de texto que contienen parte de un programa
- Dos características importantes en cualquier lenguaje
  - Encapsulamiento
    - Agrupar estructura de datos con funciones básicas
    - Un programa no es una lista de definiciones y tipos, son "cápsulas"
    - Cada una con un (tal vez más) tipo de datos y sus funciones específicas
  - Ocultamiento
    - Cuando se escribe un módulo se indica qué nombres exporta Cuáles van a poder usarse desde afuera y cuáles no
    - Aplicación
      Ocultar funciones auxiliares (como `uniónAux`)
      También para crear tipos de datos abstractos: Ocultar la representación interna

Ejemplo de módulo

- El primer ejemplo no es un tipo abstracto
- Vamos a agrupar la funcionalidad de los complejos
- Se representan bien mediante tipo algebraico
- Exportemos el tipo completo

```
module Complejos (Complejo(..), parteReal, parteIm) where

data Complejo = C Float Float

parteReal, parteIm:: Complejo -> Float

parteReal(C r i) = r

parteIm (C r i) = i
```

- `module` introduce el módulo: nombre, qué exporta
- `Tipo(..)` exporta el nombre del tipo y sus constructores
  - Permite hacer pattern matching fuera del módulo
- Después del `where` van las definiciones
- Si no hay lista de exportación se exportan todos los nombres definidos
  - `module Complejos where...`

Ejemplo de tipo abstracto

```
module Racionales (Racional, crearR, numerR, denomR)
where

data Racional = R Int Int

crearR :: Int -> Int -> Racional
crearR n d = reduce (n*signum d) (abs d)

reduce :: Int -> Int -> Racional
reduce x 0 = error "Racional con denom. 0"
reduce x y = R (x `quot` d) (y `quot` d)
    where d = gcd x y

numerR, denomR :: Racional -> Int

numerR (R n d) = n

denomR (R n d) = d
```

Aclaraciones

- signum (signo), abs (valor absoluto), quot (división entera) y gcd (gratest common divisor (MCD)) están en el preludio
- En la lista de exportación no dice (..) después de `Racional`
  - Se exporta solamente el nombre del tipo
  - NO sus constructores
  - Convierte el tipo en abstracto para los que lo usen
- Tampoco exportamos `reduce` (auxiliar)

Uso del tipo

- Volvemos al rol de usuario
  - Hay que indicar (en otro módulo) que queremos incorporar este tipo de datos
  - Se usa la cláusula `import`
    - Todos los nombres exportados por un módulo
    - O solamente algunos de ellos (aclarando entre paréntesis cuáles)

```
module Main where
    import Complejos
    import Racionales (Racional, crearR)
    miPar :: (Complejo, Racional)
    miPar = (C 1 0, crearR 4 2)
```

- Las siguientes expresiones dan error
  - `numerR (snd miPar)`
  - `reduce 4 2`
  - `R 4 2 + R 2 1`

Otra forma de crear tipos

- Vimos cómo crear sinónimos de tipos
  - Nombre adicional para un tipo existente
  - Son intercambiables
- A veces, queremos un tipo nuevo con la representación de uno existente
  - Que NO puedan intercambiarse
  - Por ejemplo, para un tipo abstracto
- Cláusula newtype

newtype

Ejemplo: conjuntos

- Los representamos internamente con listas
- Encerramos la representación en un tipo abstracto

```
newtype IntSet = Set [Int]
```

Diferencias con `data`

- Llama la atención sobre renombre
  - A otro implementador encargado de modificarla
  - A alguien que tenga que revisar el código
  - Al mismo programador dentro de un tiempo
- Admite un solo constructor con un parámetro
  - No crea nuevos elementos
  - Renombra elementos existentes (no intercambiable)
- Mejor rendimiento

Implementación de conjuntos

```
module ConjuntoInt (IntSet, vacío, esVacío, pertenece,
agregar, elegir) where

import List (insert)

newtype IntSet = Set [Int]

vacío :: IntSet
vacío = Set []

esVacío :: IntSet -> Bool
esVacío (Set xs) = null xs

pertenece :: Int -> IntSet -> Bool
pertenece x (Set xs) = x `elem` xs

agregar :: Int -> IntSet -> IntSet
agregar x (Set xs) | elem x xs = Set xs
agregar x (Set xs) | otherwise = Set (insert x xs)

elegir :: IntSet -> (Int, IntSet)
elegir (Set (x:xs)) = (x, Set xs)
```


Programación funcional

Clase 4

Terminación

Causas de indefinición

La evaluación de una función puede no terminar (bien) porque:

1. se genera una cadena infinita de reducciones:

   ```
   func1 :: Int -> Int -> Int
   func1 0 acum = acum
   func1 i acum = func1 (i-1) (2*acum)
   ```

   Si evaluamos `func1 6 5` nos devuelve 320, pero la evaluación de `func1 (-6) 5` no termina.
2. ninguna línea de la definición de la función indica qué hacer:

   ```
   func2 :: [a] -> (a,a)
   func2 [x1,x2] = (x1,x2)
   func2 (x:xs) = (x, snd (func2 xs))
   ```

   Por ejemplo, `func2 "algo1" = ('a','1')`, pero la evaluación de `func2 "a"` no termina.
3. invocamos a una función que no termina (bien)

No importa que Hugs nos devuelva el control en estos casos.
Diremos que no termina o que la función se indefine.

Terminación

Un programa (funcional) cumple con una especificación cuando para valores de entrada que satisfacen la precondición:

1. el programa devuelve un valor definido (terminación)
2. el valor devuelto satisface la poscondición (correctitud)

- en esta clase vamos a aprender un método formal para demostrar que ciertos programas terminan
- en Algoritmos y Estructuras de Datos II van a estudiar métodos para probar que los programas funcionales son correctos (inducción estructural)

Dada una función `f` y una precondición P_f, diremos que `f` termina si para cualesquiera valores de entrada $\overline{x}$ definidos que hagan verdaderos a P_f, la evaluación de `f` $\overline{x}$ termina.
Aquí, $\overline{x}$ representa a todas las variables de entrada.

Demostración de terminación

Si miramos

```
func1 :: Int -> Int -> Int
func1 0 acum = acum
func1 i acum = func1 (i-1) (2*acum)
```

puede parecer evidente que `func1` termina si tomamos $P_{\texttt{func1}} : i \geq 0$.

Veamos otro ejemplo:

```
func3 :: Int -> Bool
func3 n | n == 1 = True
        | n `mod` 2 == 0 = func3 (n `div` 2)
        | otherwise = func3 (3*n + 1)
```

¿Es evidente que `func3` termina cuando consideramos $P_{\texttt{func3}} : x > 0$?

Ecuaciones canónicas

- en Haskell hay muchas maneras de definir funciones (guardas, pattern matching, where, etc.)
- queremos usar una serie de definiciones que nos permitan analizar la función de manera homogénea
- para esto: ecuaciones canónicas

$$\begin{aligned} f\,\overline{x}/(B_1, C_1, G_1) &= E_1 \\ f\,\overline{x}/(B_2, C_2, G_2) &= E_2 \\ &\vdots \\ f\,\overline{x}/(B_n, C_n, G_n) &= E_n \end{aligned}$$

- G_i es la guarda explícita (en Haskell)
- B_i es la guarda implícita (término de tipo Bool del lenguaje de especificación). ¿Por qué entra por la i-ésima ecuación?
- C_i es el contexto de evaluación (término de tipo Bool; escrito en lenguaje de especificación) que representa información adicional proveniente de pattern matching o cláusula where.
- E_i es la expresión del lado derecho (en Haskell).

Ejemplo

problema suma(l : [Int]) = res : Int{
    asegura : $res == \sum l$;
}

```
suma :: [Int] -> Int
suma [] = 0
suma (x:xs) = x + suma xs
```

Las ecuaciones canónicas son:

$$\begin{aligned} &suma\ l\ /\ (|l| == 0, \text{true} &&, \texttt{True}) &&= 0 \\ &suma\ l\ /\ (|l| > 0\ \ , x == \text{cab}(l) \wedge xs == \text{cola}(l), \texttt{True}) &&= \texttt{x + suma xs} \end{aligned}$$

o, alternativamente,

$$\begin{aligned} &suma\ l\ /\ (|l| == 0, \text{true} &&, \texttt{True}) &&= 0 \\ &suma\ l\ /\ (|l| > 0\ \ , x == l[0] \wedge xs == l[1..|l|], \texttt{True}) &&= \texttt{x + suma xs} \end{aligned}$$

Hipótesis de la demostración

Asumimos que:

1. está demostrado que cada función g que aparece en alguna guarda o *where* termina si se utiliza cierta precondición (conocida) P_g
2. está demostrado que cada función $g \neq f$ que aparece en algún lado derecho termina si se utiliza cierta precondición (conocida) P_g

Quedan fuera de este esquema de demostración las funciones mutuamente recursivas.

Reglas de terminación

Daremos 5 reglas sobre ecuaciones canónicas de una cierta función `f`: **R1**, **R2**, **R3**, **R4** y **R5**.

- si se pueden probar las 5 reglas, entonces `f` termina
- las reglas solo predican sobre las ecuaciones canónicas; nos olvidamos de las ecuaciones originales
- si no podemos probar alguna de las 5 reglas no quiere decir que la función no termine; las 5 reglas son condiciones suficientes pero no necesarias

R1 - las funciones se evalúan dentro de su precondición

problema prLL(l : [[Int]]) = res : [$\mathbb{R}$]{
asegura res ==
[*if* $|y| > 0$ *then* $prom(y)$ *else* 0 | $y \leftarrow l$]
}

problema prom(l : [Int]) = res : $\mathbb{R}$ {
requiere $|l| > 0$;
asegura $res == prom(l)$; }
aux $prom(l : [\text{Int}])$: *Float* = $\sum l/|l|$

```
prLL :: [[Int]] -> [Float]
prLL []       = []
prLL ([]:xs) = 0 : (prLL xs)
prLL (x:xs)  = (prom x) : (prLL xs)
```

$prLL\ l\ /\ (|l| == 0$, true , `True`) = `[]`
$prLL\ l\ /\ (|l| > 0 \wedge l[0] == [\,], xs == l[1..|l|)$, `True`) = `0 : (prLL xs)`
$prLL\ l\ /\ (|l| > 0$, $l[0] == x \wedge xs == l[1..|l|)$, `True`) = `(prom x):(prLL xs)`

R1 sea i entre 1 y n, y sea g $\overline{y}$ (g puede ser `f`) una expresión que aparece en G_i o en E_i. Si

- $\overline{x}$ hace verdadero P_f,
- valen $\neg B_1, \ldots, \neg B_{i-1}$,
- si g $\overline{y}$ aparece en E_i vale además B_i,

entonces $\overline{y}$ debe hacer verdadero P_g.

En el ejemplo, `prom` siempre se invoca con argumentos que satisfacen $P_{\texttt{prom}}$.

R2 - hay suficientes ecuaciones

problema diferenciaAbsoluta(a, b : Int) = d : Int{
asegura : $d == |a - b|$;
}

```
diferenciaAbsoluta :: Int -> Int -> Int
diferenciaAbsoluta a b | a > b = a - b
                       | a < b = b - a
```

Aquí hay un problema porque falta analizar el caso $a == b$.
Debería ser, por ejemplo:

```
diferenciaAbsoluta :: Int -> Int -> Int
diferenciaAbsoluta a b | a > b = a - b
                       | a < b = b - a
                       | a == b = 0
```

R2 si $\overline{x}$ están definidas y hacen verdadero P_f, entonces existe un i entre 1 y n tal que $\overline{x}$ hacen verdadero B_i.

R3, R4 y R5 - no hay reducciones infintas

problema factorial(n : Int) = $fact$: Int{
asegura : $fact == \prod [1..n]$;
}

```
factorial :: Int -> Int -> Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

Si evaluamos `factorial 5` obtenemos 120. Pero al evaluar `factorial (-1)` se genera una cadena infinita de reducciones

Idea:

- garantizar que en cada llamado recursivo nos "acercamos" más al caso base
- dar una forma numérica de medir que tan cerca del caso base estamos
- esa forma numérica es la función variante, que será un término de tipo Int de nuestro lenguaje de especificación

R3, **R4** y **R5** - no hay reducciones infintas

$$\begin{aligned} f\,\overline{x}/(B_1, C_1, G_1) &= E_1 \\ f\,\overline{x}/(B_2, C_2, G_2) &= E_2 \\ &\vdots \\ f\,\overline{x}/(B_n, C_n, G_n) &= E_n \end{aligned}$$

Proponemos una función variante $F_v(\overline{x})$ tal que:

R3 si $\overline{x}$ están definidas y hacen verdadero P_f, entonces $F_v(\overline{x})$ no se indefine.

R4 sea i entre 1 y n, con E_i un caso recursivo, y $\overline{x}$ tales que se cumple $(P_f \wedge \neg B_1 \wedge \ldots \wedge \neg B_{i-1} \wedge B_i)$. Entonces, para cada aparición de $f\,\overline{y}$ en E_i, vale que $F_v(\overline{x}) > F_v(\overline{y})$.

R5 existe una constante entera k, que denominaremos cota de F_v, con la siguiente propiedad: toda vez que $\overline{x}$ hace verdadero P_f y $F_v(\overline{x}) \leq k$, debe existir i entre 1 y n tal que

- E_i es un caso base y
- vale $(\neg B_1 \wedge \ldots \wedge \neg B_{i-1} \wedge B_i)$.

Resumiendo

Condiciones básicas (y fáciles de demostrar)

R1 todas las funciones se evalúan dentro de su precondición

R2 hay suficientes ecuaciones (no faltan casos)

Debemos proponer una función variante $F_v(\overline{x})$ y una cota tal que (esto es más difícil de demostrar):

R3 $F_v(\overline{x})$ no se indefine (si vale P_f)

R4 $F_v(\overline{x})$ es decreciente

R5 si $F_v(\overline{x})$ pasa la cota entonces f entra a un caso base (y por lo tanto termina)

ultimo - ejemplo completo

problema ultimo($l : [T]$) $= u : T$ {
  requiere : $|l| > 0$;
  asegura : $u == l[|l| - 1]$;
}

```
ultimo :: [a] -> a
ultimo [x] = x
ultimo (x:xs) = ultimo xs
```

Las ecuaciones canónicas son:

$$\begin{aligned} &\textit{ultimo}\ l\ /\ (|l| == 1,\ x == l[0] &&,\ \texttt{True}) &&= \texttt{x} \\ &\textit{ultimo}\ l\ /\ (|l| > 0\ \ ,\ x == l[0] \wedge xs == l[1..|l|),\ \texttt{True}) &&= \texttt{ultimo xs} \end{aligned}$$

ultimo - **R1**

$$\begin{aligned} &\textit{ultimo}\ l\ /\ (|l| == 1,\ x == l[0] &&,\ \texttt{True}) &&= \texttt{x} \\ &\textit{ultimo}\ l\ /\ (|l| > 0\ \ ,\ x == l[0] \wedge xs == l[1..|l|),\ \texttt{True}) &&= \texttt{ultimo xs} \end{aligned}$$

R1 sea i entre 1 y n, y sea g $\overline{y}$ (g puede ser `f`) una expresión que aparece en G_i o en E_i. Si

- $\overline{x}$ hace verdadero P_f,
- valen $\neg B_1, \ldots, \neg B_{i-1}$,
- si g $\overline{y}$ aparece en E_i vale además B_i,

entonces $\overline{y}$ debe hacer verdadero P_g.

La única función utilizada es `ultimo`, en E_2. Tenemos que ver que si:

a) l satisface $P_{\texttt{ultimo}}$, es decir $|l| > 0$,

b) vale $\neg B_1$, esto es, $\neg |l| == 1$, y

c) vale además B_2, es decir $|l| > 0$,

entonces xs satisface $P_{\texttt{ultimo}}$, o sea: $|xs| > 0$.

- de a) y b) sabemos que $|l| \geq 2$
- de C_2 sabemos $xs == l[1..|l|)$; entonces $|xs| == |l| - 1$
- concluimos $|xs| > 0$

ultimo - R2

$$\begin{array}{lll} \textit{ultimo}\ l\ /\ (|l| == 1, & x == l[0] & , \texttt{True}) = \texttt{x} \\ \textit{ultimo}\ l\ /\ (|l| > 0 & , x == l[0] \wedge xs == l[1..|l|) & , \texttt{True}) = \texttt{ultimo xs} \end{array}$$

R2 si $\overline{\texttt{x}}$ están definidas y hacen verdadero P_f, entonces existe un i entre 1 y n tal que $\overline{\texttt{x}}$ hacen verdadero B_i.

Es claro que B_2 cumple la regla.

ultimo - R3

$$\begin{array}{lll} \textit{ultimo}\ l\ /\ (|l| == 1, & x == l[0] & , \texttt{True}) = \texttt{x} \\ \textit{ultimo}\ l\ /\ (|l| > 0 & , x == l[0] \wedge xs == l[1..|l|) & , \texttt{True}) = \texttt{ultimo xs} \end{array}$$

Proponemos $F_v(\overline{l}) = |l|$ y cota $k = 1$.

R3 si $\overline{x}$ están definidas y hacen verdadero P_f, entonces $F_v(\overline{x})$ no está indefinido.

La operación de longitud es una función total sobre listas. Es decir, no se indefine para ningún valor definido.

ultimo - R4

$$\begin{array}{lll} \textit{ultimo}\ l\ /\ (|l| == 1, & x == l[0] & , \texttt{True}) = \texttt{x} \\ \textit{ultimo}\ l\ /\ (|l| > 0 & , x == l[0] \wedge xs == l[1..|l|) & , \texttt{True}) = \texttt{ultimo xs} \end{array}$$

R4 sea i entre 1 y n, con E_i un caso recursivo, y $\overline{x}$ tales que se cumple $(\neg B_1 \wedge \ldots \wedge \neg B_{i-1} \wedge B_i)$. Entonces, para cada aparición de $f\,\overline{y}$ en E_i, vale que $F_v(\overline{x}) > F_v(\overline{y})$.

- solo tenemos un caso recursivo, que corresponde a $i = 2$
- supongamos que l cumple $(P_{ultimo} \wedge \neg B_1 \wedge B_2)$. Debemos comprobar que vale

$$F_v(l) > F_v(xs)$$

- por (el contexto de) B_2, sabemos que $xs == l[1..|l|)$, con lo cual $|xs| == |l| - 1$. Entonces

$$F_v(l) == |l| > |l| - 1 == |xs| == F_v(xs).$$

ultimo - R5

$$\begin{array}{lll} \textit{ultimo}\ l\ /\ (|l| == 1, & x == l[0] & , \texttt{True}) = \texttt{x} \\ \textit{ultimo}\ l\ /\ (|l| > 0 & , x == l[0] \wedge xs == l[1..|l|) & , \texttt{True}) = \texttt{ultimo xs} \end{array}$$

R5 toda vez que $\overline{x}$ hace verdadero P_f y $F_v(\overline{x}) \leq k$, debe existir i entre 1 y n tal que E_i es un caso base, y vale $(\neg B_1 \wedge \ldots \wedge \neg B_{i-1} \wedge B_i)$.

Debemos garantizar la entrada a un caso base cuando F_v está por debajo de la cota.

- supongamos l tal que valen P_{ultimo} y $F_v(l) \leq k$
- entonces $|l| > 0$ y $|l| \leq 1$
- como $|l|$ es un entero, debe ser necesariamente $|l| == 1$
- esto coincide con B_1, que es la guarda correspondiente al caso base $(i = 1)$.

Entonces *ultimo* termina

Como pudimos probar las reglas **R1**, **R2**, **R3**, **R4** y **R5**. concluimos que *ultimo* termina.

Observar que:

- se debe proponer una función variante F_v y una cota k para probar **R3**, **R4** y **R5**
- esa función variante y cota deben ser las mismas para probar **R3**, **R4** y **R5**
- hubiera servido una cota menor, por ejemplo, $k = 0$ o $k = -1$. ¿Por qué?

mezclarOrdenado - ejemplo completo

problema mezclarOrdenado($l1, l2 : [Int]$) $= r : [Int]\{$
requiere : $ordenado(l1) \wedge ordenado(l2)$;
asegura : $ordenado(r) \wedge mismos(r, l1 + +l2)$;
$\}$ *aux* $ordenado(l : [T]) : Bool = (\forall i \leftarrow [0..|l| - 1))\ l[i] \leq l[i + 1]$

```
mezOrd :: [Int] -> [Int] -> [Int]
mezOrd [] l2 = l2
mezOrd l1 [] = l1
mezOrd (x:xs) (y:ys) | x<=y = x : (mezOrd xs (y:ys))
mezOrd (x:xs) (y:ys) | otherwise = y : (mezOrd (x:xs) ys)
```

mezOrd $l1\ l2$ / $(|l1| == 0, true, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, true, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0 \wedge x \leq y,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{x<=y}) = \texttt{x : (mezOrd xs (y:ys))}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{True}) = \texttt{y : (mezOrd (x:xs) ys)}$

mezclarOrdenado - **R1**

mezOrd $l1\ l2$ / $(|l1| == 0, true, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, true, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0 \wedge x \leq y,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{x<=y}) = \texttt{x : (mezOrd xs (y:ys))}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{True}) = \texttt{y : (mezOrd (x:xs) ys)}$

R1 sea i entre 1 y n, y sea g $\overline{y}$ (g puede ser `f`) una expresión que aparece en G_i o en E_i. Si

- $\overline{x}$ hace verdadero P_f,
- valen $\neg B_1, \ldots, \neg B_{i-1}$,
- si g $\overline{y}$ aparece en E_i vale además B_i,

entonces $\overline{y}$ debe hacer verdadero P_g.

Observar que `mezord` siempre se llama con parámetros que verifican la precondición (las dos listas que recibe están ordenadas).

mezclarOrdenado - **R2**

mezOrd $l1\ l2$ / $(|l1| == 0, true, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, true, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0 \wedge x \leq y,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{x<=y}) = \texttt{x : (mezOrd xs (y:ys))}$
mezOrd $l1\ l2$ / $(|l1| > 0 \wedge |l2| > 0,$
$x == l1[0] \wedge xs == cola(l1) \wedge y == l2[0] \wedge ys == cola(l2),$
$\texttt{True}) = \texttt{y : (mezOrd (x:xs) ys)}$

R2 si $\overline{x}$ están definidas y hacen verdadero P_f, entonces existe un i entre 1 y n tal que $\overline{x}$ hacen verdadero B_i.

Observar que siempre ocurre que

$$|l1| = 0 \vee |l2| = 0 \vee (|l1| > 0 \wedge |l2| > 0)$$

mezclarOrdenado - Función variante y cota

mezOrd $l1\ l2$ / $(|l1| == 0, \mathit{true}, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, \mathit{true}, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0 \land x \leq y,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`x<=y) = x : (mezOrd xs (y:ys))`
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`True) = y : (mezOrd (x:xs) ys)`

Posibilidades:

- $Fv(l1, l2) == |l1|$
  - en E_3 decrece
  - pero no sirve porque en E_4 no decrece
- $Fv(l1, l2) == |l2|$
  - en E_4 decrece
  - pero no sirve porque en E_3 no decrece
- $Fv(l1, l2) == |l1| + |l2|$ es un buen candidato
- proponemos cota $k = 0$

mezclarOrdenado - **R3**

mezOrd $l1\ l2$ / $(|l1| == 0, \mathit{true}, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, \mathit{true}, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0 \land x \leq y,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`x<=y) = x : (mezOrd xs (y:ys))`
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`True) = y : (mezOrd (x:xs) ys)`

R3 si $\overline{x}$ están definidas y hacen verdadero P_f, entonces $F_v(\overline{x})$ no está indefinido.

$$Fv(l1, l2) == |l1| + |l2|$$

La operación de longitud y la suma son funciones totales.

mezclarOrdenado - **R4**

mezOrd $l1\ l2$ / $(|l1| == 0, \mathit{true}, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, \mathit{true}, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0 \land x \leq y,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`x<=y) = x : (mezOrd xs (y:ys))`
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`True) = y : (mezOrd (x:xs) ys)`

R4 sea i entre 1 y n, con E_i un caso recursivo, y $\overline{x}$ tales que se cumple $(P_f \land \neg B_1 \land \ldots \land \neg B_{i-1} \land B_i)$. Entonces, para cada aparición de $f\,\overline{y}$ en E_i, vale que $F_v(\overline{x}) > F_v(\overline{y})$.

- hay dos casos recursivos, $i = 3$ e $i = 4$
- para E_3, tenemos

$$Fv(xs, l2) == |xs| + |l2| < |l1| + |l2| = Fv(l1, l2)$$

- para E_4, tenemos

$$Fv(l1, ys) == |l1| + |ys| < |l1| + |l2| = Fv(l1, l2)$$

mezclarOrdenado - **R5**

mezOrd $l1\ l2$ / $(|l1| == 0, \mathit{true}, \texttt{True}) = \texttt{l2}$
mezOrd $l1\ l2$ / $(|l2| == 0, \mathit{true}, \texttt{True}) = \texttt{l1}$
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0 \land x \leq y,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`x<=y) = x : (mezOrd xs (y:ys))`
mezOrd $l1\ l2$ / $(|l1| > 0 \land |l2| > 0,$
$x == l1[0] \land xs == cola(l1) \land y == l2[0] \land ys == cola(l2),$
`True) = y : (mezOrd (x:xs) ys)`

R5 toda vez que $\overline{x}$ hace verdadero P_f y $F_v(\overline{x}) \leq k$, debe existir i entre 1 y n tal que E_i es un caso base, y vale $(\neg B_1 \land \ldots \land \neg B_{i-1} \land B_i)$.

Debemos garantizar la entrada a un caso base cuando $F_v(l1, l2) \leq 0$.

- supongamos $Fv(l1, l2) = |l1| + |l2| \leq 0$
- entonces $|l1| == 0$ y $|l2| == 0$
- en particular, vale B_1 (también vale B_2)

Programación funcional

Clase 5

Correctitud - órdenes de evaluación - alto orden

Correctitud

- ya vimos como demostrar terminación
- para demostrar que una definición es correcta, hace falta
  - una especificación
    problema $fact(n : \text{Int}) = result : \text{Int}\{$
      requiere $n \geq 0$;
      asegura $result == \prod[1..n]$;
    $\}$
  - y un programa
    ```
    fact 0 = 1
    fact n | n > 0 = n * fact (n-1)
    ```
- definición recursiva: es natural demostrar por inducción

Demostración de correctitud

Quiero probar que para todo x, la función `fact` con entrada x devuelve $\prod[1..x]$.

- caso base ($n == 0$)
  - la especificación dice que hay que probar que la función `fact` con entrada 0 devuelve $\prod[1..0] == \prod[\,] = 1$
  - pero eso es justamente lo que devuelve `fact` en el caso base
- hipótesis inductiva (HI): `fact n` $== \prod[1..n]$
  - la especificación dice que hay que probar que la función `fact` con entrada $n+1$ devuelve $\prod[1..n+1]$
  - calculemos `fact n` $+ 1$
  - gracias a la precondición, $n+1 > 0$. Entonces uso la segunda ecuación instanciando en $n+1$

$$fact(n+1) == (n+1) * fact(n)$$

  - usando HI:

$$\begin{aligned} fact(n+1) &== (n+1) * \prod[1..n] \\ &== \prod[1..(n+1)] \end{aligned}$$

  - entonces, demostré que `fact` con entrada $n+1$ devuelve $\prod[1..(n+1)]$

Principio de inducción para $\mathbb{N}$

- sea $P(x)$ una propiedad de $x \in \mathbb{N}$
- supongamos que queremos probar que vale $P(x)$ para todo $x \in \mathbb{N}$
- el axioma de inducción dice:
  - si
    - $P(0)$ es verdadero y
    - para todo n, $P(n) \rightarrow P(n+1)$ es verdadero
  - entonces $P(x)$ es verdadero para todo $x \in \mathbb{N}$
- el axioma de inducción completa dice:
  - si
    - $P(0)$ es verdadero y
    - para todo n, $(\forall m \leq n\ P(m)) \rightarrow P(n+1)$ es verdadero
  - entonces $P(x)$ es verdadero para todo $x \in \mathbb{N}$
- en el ejemplo anterior, $P(x)$ dice

  "la función `fact` *con entrada* x *devuelve* $\prod[1..x]$*"*

Otro ejemplo de demostración de correctitud

- vimos cómo demostrar la correctitud de $fact(n)$ haciendo inducción en el parámetro de entrada (es decir, en n)
- también se puede hacer inducción en el valor que toma una función
- por ejemplo,

problema $suma(l : [\text{Int}]) = result : \text{Int}\{$
  asegura $result == \sum l$;
$\}$

```
suma [] = 0
suma (x:xs) = x + suma xs
```

  - quiero probar $P(l)$: "la salida de `suma` con entrada l es $\sum l$"
  - se puede hacer inducción en $|l|$ (que toma valores en $\mathbb{N}$)
    - caso base: pruebo $P(l)$ para l tal que $|l| == 0$.
      Es decir, tengo que probar $P(l)$ para $l == [\,]$.
      `suma` con entrada $[\,]$ devuelve $0 = |l|$.
    - HI: vale $P(l)$ para toda l tal que $|l| \leq n$ ($n \in \mathbb{N}$)
      Quiero ver que vale $P(l)$ para toda l tal que $|l| == n + 1$.
      Sea y tal que $|y| == n + 1$. Quiero ver que vale $P(y)$.
      Como $|y| == n + 1$, sabemos que $y == x : xs$ para alguna lista xs y entero x.
      La salida de `suma` con entrada y es $x +$ salida de `suma` con entrada xs.
      Uso la HI (notar que $|xs| \leq n$): la salida de `suma` con entrada y es $x + \sum xs$.
      Por propiedad de $\sum$, $x + \sum xs == \sum(x : xs) == \sum y$.
      Entonces la salida de `suma` con entrada y es $\sum y$.
      Es decir, vale $P(y)$.

Principio de inducción para tipos algebraicos

- se llama inducción estructural. La van a estudiar en *Algoritmos y Estructuras de Datos II*
- sea $P(x)$ una propiedad de un elemento x de un tipo algebraico T
- supongamos que queremos probar que vale $P(x)$ para todo x de tipo T
- sea c una función matemática $T \to \mathbb{N}$
- el axioma de inducción dice:
  - si
    - $P(y)$ es verdadero para toda y tal que $c(y) = 0$ y
    - para todo y, si $\forall y\ (c(y) = n \to P(y))$ entonces $\forall y\ (c(y) = n + 1 \to P(y))$
  - entonces $P(y)$ es verdadero para todo y de tipo T
- el axioma de inducción completa dice:
  - si
    - $P(y)$ es verdadero para toda y tal que $c(y) = 0$ y
    - para todo y, si $\forall y\ (c(y) \leq n \to P(y))$ entonces $\forall y\ (c(y) = n + 1 \to P(y))$
  - entonces $P(y)$ es verdadero para todo y de tipo T
- en el ejemplo anterior T es el tipo [Int] y
  - $P(y)$ dice "la función `suma` con entrada y devuelve $\sum y$"
  - $c(y) = |y|$.

Reducción

Modelo de cómputo:

- cómo se calcula el valor de una expresión
- puede afectar la semántica
  - distintos modelos pueden dar distintos resultados

Reducción:

- mecanismo de evaluación en Haskell
- reemplazar una subexpresión por otra
  - reemplazada:
    - instancia del lado izquierdo de una ecuación orientada
    - se llama redex (*reducible expression*) o radical
  - reemplazante:
    - lado derecho, instanciado de manera acorde
    - el resto de la expresión queda igual
- instanciación
  - asignación de expresiones a variables de un pattern

Ejemplo

- expresión
  `suma (restar 2 (amigos Juan)) 4`
- ecuación
  `restar x y = x - y`
- reducción
  1. busco un redex y asignación
     `suma` $\underbrace{(\texttt{restar 2 (amigos Juan)})}_{\text{redex}}$ `4`
     - asignación:
       `x` $\leftarrow$ `2`
       `y` $\leftarrow$ `(amigos Juan)`
  2. reemplazo el redex con esa asignación

`suma (restar 2 (amigos Juan)) 4` $\rightsquigarrow$ `suma (2 - (amigos Juan)) 4`

Formas normales

- valor de una expresión
  - sigo reduciendo hasta que no haya más redexes
- obtengo una forma normal
  - solamente constantes y constructores
- son formas normales
  - 2
  - `(3, True, C 5.1 (-6.2))`
- no son formas normales
  - `1 + 1`
  - `(9 ‘quot‘ 3, 3>=2, sumarC (C 1.05 (-12.4)) (C 4.05 (6.2)))`

Mecanismo de reducción

1. si la expresión está en forma normal, terminamos
2. si no, buscar un redex
3. reemplazarlo
4. volver a empezar

- todos los mecanismos de reducción comparten este algoritmo
  - las estrategias dependen del paso 2
    - en qué orden elijo los redexes
    - se las llama órdenes de reducción
  - pueden influir en que se llegue o no a una forma normal

Normalización

- ¿toda expresión tiene forma normal?
  - no. Las siguientes expresiones no tienen forma normal
    - `f x = f (f x)` ¿cuanto vale `f 3`?
    - `infinito = infinito + 1` ¿cuanto vale `infinito`?
    - `inverso x | x /= 0 = 1 / x` ¿cuanto vale `inverso 0`?
- si se consigue, ¿toda estrategia encuentra la misma?
  - sí
  - se llama confluencia

Bottom

- las expresiones que no tienen forma normal se llaman indefinidas
  - se puede decir que su valor es $\bot$
- podríamos definirlo en Haskell

```
bottom :: a
bottom = bottom
```

- cualquier intento de evaluar bottom se indefine

```
g :: Int -> Int
g x = if x == bottom then 1 else 0
```

`g 2` $\rightsquigarrow \bot$

Indefinición

- le pasamos un valor definido a una función
  - parciales: a veces devuelven $\bot$
  - totales: nunca
- le pasamos $\bot$ a una función
  - ¿devuelve $\bot$?
    - no siempre
    - depende de sus ecuaciones y del orden de reducción
  - estrictas: $f\bot \rightsquigarrow \bot$
  - no estrictas: $f\bot \rightsquigarrow$ valor

Totales vs. parciales

- total

```
suc :: Integer -> Integer
suc x = x + 1
```

- parciales

```
recip :: Float -> Float
recip x | x /= 0 = 1/x
```

- las dos son estrictas
  - si les paso $\bot$ devuelven $\bot$

Estrictas vs. no estrictas

- ecuaciones

```
const :: a -> b -> a
const x y = x
```

- ¿cuánto vale?
  - `const 2 infinito`

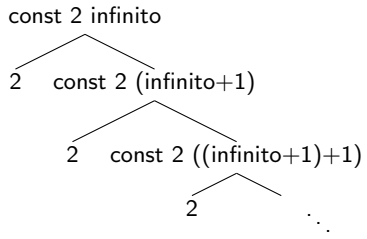
Depende del diseño del lenguaje. El secreto está en el orden de evaluación

Orden de evaluación

- forma de elegir el próximo redex
- recordar confluencia
  - si por dos órdenes llegamos a valores definidos, es el mismo valor
  - pero puede ser que un orden llegue a $\bot$ y otro no

```
infinito :: Integer -> Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```



Órdenes de evaluación

- aplicativo
  - primero redexes internos
  - primero los argumentos, después la aplicación
- normal
  - el redex más externo
    - para el que pueda hacer pattern matching
  - primero la aplicación, después los argumentos
    - si se necesitan
  - siempre encuentra la forma normal
    - si la hay
- los dos empiezan a izquierda
  - en caso de más de un redex del mismo nivel

Ejemplos de evaluación aplicativa

```
f x = 0
```

f (1/0) se indefine

```
infinito :: Integer -> Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```

const 2 infinito $\rightsquigarrow$ const 2 (infinito+1) $\rightsquigarrow$ const 2 ((infinito+1)+1) $\rightsquigarrow$
const 2 (((infinito+1)+1)+1) $\rightsquigarrow \ldots$

```
head :: [a] -> a
head (x:xs) = x
```

```
tail :: [a] -> a
tail (x:xs) = xs
```

```
inc :: [a] -> a
inc [] = []
inc (x:xs) = (x+1):inc xs
```

head (tail (inc [1,2,3,4])) $\rightsquigarrow$ head (tail (2:inc [2,3,4])) $\rightsquigarrow$
head (tail (2:3:inc [3,4])) $\rightsquigarrow$ head (tail (2:3:4:inc [4])) $\rightsquigarrow$
head (tail (2:3:4:5:inc [])) $\rightsquigarrow$ head (tail (2:3:4:5:[])) $\rightsquigarrow$ head (3:4:5:[]) $\rightsquigarrow$ 3

Ejemplos de evaluación normal

```
f x = 0
```

f (1/0) $\rightsquigarrow$ 0

```
infinito :: Integer -> Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```

const 2 infinito $\rightsquigarrow$ 2

```
head :: [a] -> a
head (x:xs) = x
```

```
tail :: [a] -> a
tail (x:xs) = xs
```

```
inc :: [a] -> a
inc [] = []
inc (x:xs) = (x+1):inc xs
```

head (tail (inc [1,2,3,4])) $\rightsquigarrow$ head (tail (2:inc [2,3,4])) $\rightsquigarrow$ head (inc [2,3,4]) $\rightsquigarrow$ head (3:inc [3,4]) $\rightsquigarrow$ 3

Propiedades de los órdenes

- orden aplicativo
  - `const` es estricta
  - todas son estrictas
- orden normal
  - `const` es no estricta
  - hay funciones estrictas y no estrictas
  - depende de si necesitan evaluar todos sus argumentos

Órdenes y performance

```
fib 1 = 1
fib 2 = 1
fib n | n > 2 = fib (n-1) + fib (n-2)
```

¿Cuántas reducciones necesito?

- `fib 20` $\longrightarrow$ 200.000
- `const 3 (fib 20)` $\longrightarrow$?
  - aplicativo: 200.001
  - normal: 1
- `quin x = x + x + x + x + x`
  `quin (fib 20)` $\longrightarrow$?
  - aplicativo: 200.005
  - normal: (fib 20) + (fib 20) + (fib 20) + (fib 20) + (fib 20) $\longrightarrow$ 1.000.000

Evaluación *lazy*

- perezosa, haragana, fiaca
- algoritmo usado en Haskell
- orden normal, pero...
  - aprovecha la transparencia referencial
  - si una expresión vuelve a aparecer, se acuerda el valor anterior
  - `quin (fib 20)` $\longrightarrow$ 200.005

Estructuras infinitas

- es una ventaja de la evaluación lazy
- ejemplos
  - ```
    naturales, pares, impares :: [Integer]
    naturales = secuencia 0 1
    impares = secuencia 1 2
    pares = secuencia 0 2

    secuencia :: Integer -> Integer -> [Integer]
    secuencia n p = n:secuencia (n+p) p
    ```
- evaluación de estructuras infinitas
  - `head naturales` $\rightsquigarrow$ `head (secuencia 0 1)` $\rightsquigarrow$ `head (0:secuencia 1 1)` $\rightsquigarrow$ `0`
  - `take 10 impares` $\rightsquigarrow \cdots \rightsquigarrow$ `[1,3,5,7,9,11,13,15,17,19]`

Alto orden

- filtrar los elementos mayores a 2 de una lista:
  ```
  filtroMayoresADos :: [Int] -> [Int]
  filtroMayoresADos [] = []
  filtroMayoresADos (x:xs) | x>2 = x:filtroMayoresADos xs
                           | otherwise = filtroMayoresADos xs
  ```
- filtrar los elementos pares de una lista:
  ```
  filtroPares :: [Int] -> [Int]
  filtroPares [] = []
  filtroPares (x:xs) | (x `mod` 2==0) = x:filtroPares xs
                     | otherwise = filtroPares xs
  ```
- más general:
  ```
  filter: (Int -> Bool) -> [Int] -> [Int]
  filter f [] = []
  filter f (x:xs) | f x = x:filter f xs
                  | otherwise = filter f xs
  ```
  - ```
    filtroMayoresADos xs = filter mayorADos xs
                           where mayorADos x = x>2
    ```
  - ```
    filtroPares xs = filter par xs
                     where par x = (x `mod` 2 == 0)
    ```
- las funciones de alto orden son aquellas que reciben o devuelven funciones

Alto orden

- sumar los elementos de una lista
  ```
  suma [] = 0
  suma (n:ns) = n + suma ns
  ```
- verificar que sean todos verdaderos
  ```
  all [] = True
  all (b:bs) = b && all bs
  ```
- aplanar una lista de listas
  ```
  concat [] = []
  concat (xs:xss) = xs ++ concat xss
  ```
- más general:
  ```
  foldr f a [] = a
  foldr f a (x:xs) = x `f` (foldr f a xs)
  ```
  - `suma = foldr (+) 0`
  - `all = foldr (&&) True`
  - `concat = foldr (++) []`
- ¿cuál es el tipo de `foldr`?
  - de los ejemplos se deduce `(a -> a -> a) -> a -> [a] -> a`
  - pero puede ser un poco más general:

    `(a -> b -> b) -> b -> [a] -> b`

Currificación

- ```
  suma' :: (Int , Int) -> Int
  suma' (x,y) = x + y
  ```
  - recibe un par ordenado
- ```
  suma :: Int -> Int -> Int
  suma x y = x + y
  ```
  - el tipo de `suma` se interpreta como `suma :: Int -> (Int -> Int)`
  - es una función que recibe un entero y devuelve una función de enteros en enteros
  - ¿qué es `suma 1`?
    - una función de tipo `Int -> Int`
    - `(suma 1) y = suma 1 y = 1+y`
  - ¿qué es `suma 23`?
    - también es una función de tipo `Int -> Int`
    - `(suma 23) y = suma 23 y = 23+y`
  - en general `suma x y = (suma x) y = x+y`
  ```

Programación imperativa

Clase 1

Introducción a la programación imperativa

Programación imperativa (diferencias con funcional)

- los programas no necesariamente son funciones
  - ahora pueden *devolver* más de un valor
  - hay nuevas formas de pasar argumentos
- nuevo concepto de variables
  - posiciones de memoria
  - cambian explícitamente de valor a lo largo de la ejecución de un programa
  - pérdida de la transparencia referencial
- nueva operación: la asignación
  - cambiar el valor de una variable
- las funciones no pertenecen a un tipo de datos
- distinto mecanismo de repetición
  - en lugar de la recursión usamos la iteración
- nuevo tipo de datos: el arreglo
  - secuencia de valores de un tipo (como las listas)
  - longitud prefijada
  - acceso directo a una posición (en las listas, hay que acceder primero a las anteriores)

Lenguaje C++

- vamos a usarlo para la programación imperativa (también soporta parte del paradigma de objetos)
- vamos a usar un subconjunto (como hicimos con Haskell)
  - no objetos, no memoria dinámica, etc.
  - sí vamos a usar la notación de clases, para definir tipos de datos
- el tipado es más débil que el de Haskell
- tipos básicos:
  - `char`
  - `float`
  - `int`

  (con minúscula)

Programa en C++

- colección de tipos y funciones
- definición de función

  tipoResultado nombreFunción (parámetros)
  bloqueInstrucciones

- ejemplo

```
int suma2 (int x, int y) {
  int res = x + y;
  return res;
}
```

- su evaluación consiste en ejecutar una por una las instrucciones del bloque
- el orden entre las instrucciones es importante
  - siempre de arriba hacia abajo

Variables en imperativo

- nombre asociado a un espacio de memoria
- puede cambiar de valor varias veces en la ejecución
- en C++ se declaran dando su tipo y su nombre
  - `int x;` $\longrightarrow$ x es una variable de tipo int
  - `char c;` $\longrightarrow$ c es una variable de tipo char
- programación imperativa
  - conjunto de variables
  - instrucciones que van cambiando sus valores
  - los valores finales, deberían resolver el problema

La asignación

- operación fundamental para modificar el valor de una variable
- sintaxis

  variable = *expresión;*
- operación asimétrica
  - lado izquierdo: debe ir una variable u otra expresión que represente una posición de memoria
  - lado derecho: puede ser una expresión del mismo tipo que la variable
    - constante
    - variable
    - función aplicada a argumentos
- efecto de la asignación
  1. se evalúa el valor de la expresión de la derecha
  2. ese valor se copia en el espacio de memoria de la variable
  3. el resto de la memoria no cambia
- ejemplos: `x = 0;` `y = x;` `x = x+x;`
  `x = suma2(z+1,3);` `x = x*x + 2*y + z;`
- no son asignaciones:
  `3 = x;` `doble (x) = y;` `8*x = 8;`

La instrucción *return*

- termina la ejecución de una función
- retorna el control a su invocador
- devuelve la expresión como resultado

```
int suma2 (int x, int y) {
  int res = x + y;
  return res;
}
```

```
int suma2 (int x, int y) {
  return x + y;
}
```


Transformación de estados

- llamamos estado de un programa a los valores de todas sus variables en un punto de su ejecución
  - antes de ejecutar la primera instrucción
  - entre dos instrucciones
  - después de ejecutar la última instrucción
- podemos ver una ejecución de un programa como una sucesión de estados
- la asignación es la instrucción que transforma estados
- el resto de las instrucciones son de control
  - modifican el flujo de ejecución
  - orden de ejecución de las instrucciones

Afirmaciones en imperativo

- al demostrar una propiedad, agregamos afirmaciones sobre el estado entre instrucciones
- se amplía el lenguaje de especificación con la sentencia vale

  vale nombre: P;

- el nombre (*nombre*) es opcional
- *P* es un predicado
  - expresión de tipo Bool del lenguaje de especificación
- se coloca entre dos instrucciones
- significa que *P* vale en ese punto del programa en cualquier ejecución
- el compilador no entiende las sentencias del lenguaje de especificación
- para que no dé error, las ponemos en comentarios
  - como en Haskell, pero con otra sintaxis
  - hay dos maneras:
    - `//` comenta una sola línea
    - `/*...*/` comenta varias líneas

Cláusulas *vale* y *estado*

- ejemplo de código con afirmaciones

```
x = 0;
  //vale x == 0;
x = x + 3;
  //vale x == 3;
x = 2 * x;
  //vale x == 6;
```

- para referirnos al valor que tenía en un estado anterior
- usamos estado para dar un nombre al estado
- nos referimos al estado anterior con

  variable@nombreEstado

- semántica de la asignación:

```
  //estado a
v = e;
  //vale v == e@a
```

- después de ejecutar la asignación, la variable *v* tiene el valor que tenía antes la expresión *e*
- en cada ejecución el estado puede ser distinto

Cláusula *pre*

- el operador pre se refiere al estado previo a la ejecución de una función

  pre(*expresión*)

- representa el valor de la expresión en el estado inicial
- cuando los parámetros de una función reciben el valor de los argumentos con los que se la invocó

problema *suc*(*x* : Int) = *res* : Int {
  asegura *res* == *x* + 1; }

```
int suc(int x) {
  int y = x;
    //estado a
    //vale y == x;
  y = y + 1;
    //vale y == y@a + 1;
  return y;
  }
```

```
int suc(int x) {
    //local x
    //vale x == pre(x)
  x = x + 1;
    //vale x == pre(x) + 1;
  return x;
  }
```


Cláusula *local*

- en las afirmaciones, se presupone que los parámetros mantienen su valor inicial
- en cualquier estado *n*

  vale parámetro@n == pre*(parámetro)*

- evita tener que repetir esta afirmación en cada estado, excepto
  - parámetros que aparecen en cláusulas modifica de la especificación
  - si queremos usar un parámetro como variable temporaria, usamos la cláusula local

problema *suma1*(*x*, *y* : Int) {
  modifica *x*;
  asegura *x* == pre(*x*) + *y*; }

```
void suma1(int &x, int y) {
    //estado a
    //vale x == pre(x);
    //vale y == pre(y); (no hace falta)
  x = x + y;
    //vale x == pre(x) + y;
    //vale y == pre(y); (no hace falta)
  }
```

problema *suma2*(*x*, *y* : Int) = *res* : Int {
  asegura *res* == *x* + *y*; }

```
int suma2(int x, int y) {
    //local x
    //vale x == pre(x);
    //vale y == pre(y); (no hace falta)
  x = x + y;
    //vale x == pre(x) + y;
    //vale y == pre(y); (no hace falta)
  return x;
  }
```

Cláusula *implica*

- cláusula

  implica P;

- se pone después de una o más afirmaciones *vale* o *implica*
- indica que P se deduce de las afirmaciones anteriores

problema $suc(x : \text{Int}) = res : \text{Int}$ {
  asegura $res == x + 1$; }

¡todas las cláusulas *implica* deben ser justificadas!

- en el ejemplo,
  - //implica $x == \text{pre}(x) + 2 - 1$: sale de reemplazar $x@a$ por $\text{pre}(x) + 2$, de acuerdo a lo que dice el estado a
  - //implica $x == \text{pre}(x) + 1$: operaciones aritméticas
  - //implica $res == \text{pre}(x) + 1$: sale de reemplazar $x@b$ por $\text{pre}(x) + 1$, de acuerdo a lo que dice el estado b

```
int suc(int x) {
    //local x
  x = x + 2;
    //estado a
    //vale x == pre(x) + 2;
  x = x - 1;
    //estado b
    //vale x == x@a - 1;
    //implica x == pre(x) + 2 - 1;
    //implica x == pre(x) + 1;
  return x;
    //vale res == x@b;
    //implica res == pre(x) + 1;
}
```


Pasaje de argumentos

- los argumentos tienen que pasar del invocador al invocado
- hay que establecer una relación entre argumentos y parámetros
- las convenciones más habituales son dos
  - por valor
    - antes de hacer la llamada se obtiene el valor del argumento
    - se lo coloca en la posición de memoria del parámetro correspondiente
    - durante la ejecución de la función, se usan esas copias de los argumentos
    - por eso también se llama por copia
    - los valores originales quedan protegidos contra escritura
  - por referencia
    - en la memoria asociada a un parámetro se almacena la dirección de memoria del argumento correspondiente
    - el parámetro se convierte en una referencia indirecta al argumento
    - todas las asignaciones al parámetro afectan directamente el valor de la variable pasada como argumento
    - el argumento no puede ser una expresión cualquiera (por ejemplo, un literal) debe indicar un espacio de memoria (en general, es una variable)

Referencias en C++

- referencia a una variable
- constructor de tipos &
- actúa como un alias (se usa sin sintaxis especial, como el tipo original)

```
int &b = a;
b = 3;
  //vale a == b == 3;
a = 4;
  //vale a == b == 4;
```

- complica la comprensión del programa y su semántica
- por eso pedimos que una posición de memoria no tenga dos nombres

Pasaje de argumentos en C++

- siempre por valor
- para pasar por referencia, se usan referencias
- también se pasan por valor (copia), pero lo que se copia no es el valor del argumento, se copia su dirección
- la modificación del parámetro implica modificación del argumento
- indicamos qué parámetros son de tipo referencia con el operador & (pueden usarse como parámetros de salida o de entrada/salida)

```
void A(int &i) {
  i = i-1;
}
```

```
void B(int i) {
  i = i-1;
}
```

```
void C() {
  int j = 6;
    //vale j == 6;
  A(j);
    //vale j == 5;
  B(j);
    //vale j == 5;
}
```

- no se puede pasar a A una expresión que no sea una variable (u otra descripción de una posición de memoria)

Pasaje por referencia para las variables en *modifica*

- en el lenguaje de especificación sabemos indicar que los argumentos se modifican

  problema swap(x, y : Int) {
  modifica x, y;
  asegura $x == \text{pre}(y) \wedge y == \text{pre}(x)$; }

- los lenguajes imperativos permiten implementar esto directamente

```
void swap(int& x, int& y) {
  int z;      //estado 1; vale x == pre(x) ∧ y == pre(y);
  z = x;      //estado 2; vale z == x@1 ∧ x == x@1 ∧ y == y@1;
  x = y;      //estado 3; vale z == z@2 ∧ x == y@2 ∧ y == y@2;
  y = z;      //estado 4; vale z == z@3 ∧ x == x@3 ∧ y == z@3;
    //implica x == pre(y) ∧ y == pre(x);
}
```

- justificación del *implica*: $x == x@3 == y@2 == y@1 == \text{pre}(y)$; $y == z@3 == z@2 == z@1 == \text{pre}(x)$
- el estado 4 cumple la poscondición (entonces, la función es correcta respecto de su especificación)

¿Qué hace esta función?

```
void prueba(int &x, int &y) {
    //estado 1; vale x == pre(x) ∧ y == pre(y);
  x = x + y;
    //estado 2; vale y == y@1 ∧ x == x@1 + y@1;
    //implica x == pre(x) + pre(y);  (pues y@1 == pre(y) y x@1 == pre(x))
  y = x - y;
    //estado 3; vale x == pre(x) + pre(y) ∧ y == x@2 − y@2;
    //implica y == pre(x) + pre(y) − pre(y);  (pues y@2 == y@1 == pre(y))
    //implica y == pre(x);  (operaciones aritméticas)
  x = x - y;
    //estado 4; vale y == pre(x) ∧ x == x@3 − y@3;
    //implica x == pre(x) + pre(y) − pre(x);  (justificación trivial)
    //implica y == pre(x) ∧ x == pre(y);  (operaciones aritméticas)
}
```

por lo tanto, esta función también es correcta respecto de la especificación del problema swap

- salvo que sea llamada con `prueba(x,x)`

Invocación de funciones - pasaje por valor

problema doble(x : Int) = *res* : Int {
asegura *res* $== 2 * x$; }

problema cuad(x : Int) = *res* : Int {
asegura *res* $= 4 * x$; }

```
int cuad(int x) {
  int c = doble(x);
     //estado 1;
     //vale c == 2 * x; (usando la poscondición de doble)
  c = doble(c);
     //vale c == 2 * c@1; (usando la poscondición de doble)
     //implica c == 2 * 2 * x == 4 * x;  (justificación trivial)
  return c;
     //vale res == 4 * x;
  }
```


Invocación de funciones - pasaje por referencia

problema swap(x, y : Int) {
modifica x, y;
asegura $x == \text{pre}(y) \wedge y == \text{pre}(x)$; }

problema swap3(a, b, c : Int) {
modifica a, b, c;
asegura $a == \text{pre}(b) \wedge b == \text{pre}(c) \wedge c == \text{pre}(a)$; }

```
void swap3(int& a, int& b, int& c) {
    //vale a == pre(a) ∧ b == pre(b) ∧ c == pre(c);
  swap(a,b);
    //estado 1;
    //vale a == pre(b) ∧ b == pre(a); (usando la poscondición de swap)
    //vale c == pre(c);
  swap(b,c) ;
    //estado 2;
    //vale b == c@1 ∧ c == b@1; (usando la poscondición de swap)
    //vale a == a@1;
    //implica a == pre(b) ∧ b == pre(c) ∧ c == pre(a);  (justificar...)
  }
```

Programación imperativa

Clase 2

Teorema del Invariante

Asignación

Semántica de la asignación:

- sea e una expresión cuya evaluación no modifica el estado

```
//estado a
v = e;
//vale v == e@a ∧ z1 = z1@a ∧ ··· ∧ zk = zk@a
```

  - donde $z_1, \ldots, z_k$ son todas las variables del programa en cuestión distintas a v que aparezcan en una cláusula *modifica* o *local*
    - las otras variables se supone que no cambian así que no hace falta decir nada)
- si la expresión e es la invocación a una función que recibe parámetros por referencia, puede haber más cambios, pero al menos

```
//vale v == e@a
```

está en la poscondición de la asignación

Condicionales

- supongamos este código:

```
if (B) uno else dos;
```

  - B tiene que ser una expresión booleana (verdadera o falsa) sin efectos secundarios (no tiene que modificar el estado)
    - se llama guarda
  - `uno` y `dos` son instrucciones
    - en particular, pueden ser bloques (entre llaves)
- si sabemos que

```
//vale P ∧ B
uno;
//vale Q
```

```
//vale P ∧ ¬B
dos;
//vale Q
```

- entonces podemos afirmar

```
//vale P
if (B) uno else dos;
//vale Q
```

- decimos que P es la precondición del condicional y Q es su poscondición

Condicionales

Equivalentemente:

- si sabemos que

```
//vale P ∧ B
uno;
//vale Q1
```

```
//vale P ∧ ¬B
dos;
//vale Q2
```

- entonces podemos afirmar

```
//vale P
if (B) uno else dos;
//vale Q1 ∨ Q2
```

Ejemplo de demostración de condicional

problema $max(x, y : \text{Int}) = result : \text{Int}\{$
  asegura $Q : (x > y \land result == x) \lor (x \leq y \land result == y)$
$\}$

```
int max(int x, int y) {
  int m = 0;
    //vale P_if : m == 0;
  if (x > y)
      m = x;
  else
      m = y;
    //vale Q_if : (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);
  return m;
    //vale Q;
}
```


Ejemplo de demostración de condicional

- demuestro cada rama fuera del condicional
  - rama true:
    //vale $m == 0 \land x > y$;
    `m = x;`
    //vale $x > y \land m == x$;
    //implica $(x > y \land m == x) \lor (x \leq y \land m == y)$;
    (justificación: $p \rightarrow (p \lor q)$ es tautología)
  - rama false:
    //vale $m == 0 \land x \leq y$;
    `m = y;`
    //vale $x \leq y \land m == y$;
    //implica $(x > y \land m == x) \lor (x \leq y \land m == y)$;
    (justificación: $p \rightarrow (p \lor q)$ es tautología)
- pudimos llegar a Q_{if} por las dos ramas,
  - entonces demostré que el condicional es correcto para la precondición P_{if} y poscondición Q_{if}

Ciclos

- `while (B) cuerpo;`
- B: expresión booleana, sin efectos colaterales
  - también se la llama guarda
- cuerpo es una instrucción
  - en particular, puede ser un bloque (entre llaves)
  - se repite mientras B valga
    - cero o más veces
    - si es una cantidad finita, el programa termina
    - si es > 0, alguna de las ejecuciones tiene que hacer B falsa
    - y el estado final de esa ejecución será el estado final del ciclo

Ejemplo de ciclo

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

- en imperativo

```
int sumat (int x) {
    int s = 0, i = 0;
    while (i < x) {
  // estado 1
        i = i + 1;
        s = s + i;
  // estado 2
    }
    return s;
}
```

- en funcional

```
sumat :: Int -> Int
sumat 0 = 0
sumat n = n + (sumat (n-1))
```

estados para $x == 4$

i@1	s@1	i@2	s@2
0	0	1	1
1	1	2	3
2	3	3	6
3	6	4	10

Observar que en cada paso:

- $0 \leq i \leq x$
  - cuando $i == x$, sale del ciclo
- $s == \sum[0..i]$

Estas dos valen en estado 1 y estado 2 (pero no en el medio)

Otro ejemplo de ciclo

problema $fact(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \prod[1..x]$
$\}$

- en imperativo

```
int fact (int x) {
    int f = 1, i = 0;
    while (i < x) {
  // estado 1
        i = i + 1;
        f = f * i;
  // estado 2
    }
    return f;
}
```

- en funcional

```
fact :: Int -> Int
fact 0 = 1
fact n = n * (fact (n-1))
```

estados para $x == 4$

i@1	f@1	i@2	f@2
0	1	1	1
1	1	2	2
2	2	3	6
3	6	4	24

Observar que en cada paso:

- $0 \leq i \leq x$
  - cuando $i == x$, sale del ciclo
- $f == \prod[1..i]$

Estas dos valen en estado 1 y estado 2 (pero no en el medio)

Semántica de ciclos

- la semántica requiere cuatro expresiones del lenguaje de especificación
  - una precondición P_C
  - una poscondición Q_C
  - un invariante I
  - una guarda B
- invariante
  - condición cuya veracidad es preservada por el cuerpo del ciclo
  - vale antes de entrar al ciclo (justo antes de evaluar la guarda por primera vez)
  - vale en cada iteración
    - justo después de entrar al cuerpo del ciclo
    - y justo después de ejecutar la última instrucción del cuerpo del ciclo
    - pero pude no valer en el medio del cuerpo

```
  //vale P_C;
while (B) {
  //vale I ∧ B;
  cuerpo
  //vale I;
}
  //vale Q_C;
```

  - no hay forma algorítmica de encontrarlo
  - conviene darlo al escribir el ciclo, porque encierra la idea del programador o diseñador

Terminación y correctitud

```
  //vale P_C;
while (B) {
  //vale I ∧ B;
  cuerpo
  //vale I;
}
  //vale Q_C;
```

Herramientas:

- P_C = precondición del ciclo
- Q_C = poscondición del ciclo
- I invariante
- B guarda

Especificación del ciclo = (P_C, Q_C)

- un ciclo termina (con respecto a la especificación del ciclo) si no importa para qué valores de entrada que satisfagan P_C, el ciclo, después de una cantidad finita de pasos termina (es decir, se verifica $\neg B$)
  - para demostrar esto, necesitamos herramientas adicionales:
    - función variante
    - cota
- un ciclo es correcto (con respecto a la especificación del ciclo) si termina y al salir satisface Q_C
  - para demostrar esto, vamos a usar fuertemente el invariante

Terminación

- ¿cómo podemos garantizar que el ciclo termina?
- similares a conceptos que vimos para programación funcional

```
  //vale P_C;
while (B) {
  //vale I ∧ B;
  cuerpo
  //vale I;
}
  //vale Q_C;
```

- para hablar de la cantidad de veces que se ejecuta un ciclo necesitamos:
  - expresión variante (v)
    - es una expresión del lenguaje de especificación, de tipo Int
    - usa variables del programa
    - debe decrecer en cada paso

```
//estado a;
//vale B ∧ I;
cuerpo
//vale v < v@a;
```

  - cota (c)
    - valor (fijo, por ejemplo 0 o -8) que, si es alcanzado o pasado por la expresión variante, garantiza que la ejecución sale del ciclo
    - más formalmente: $(I \wedge v \leq c) \rightarrow \neg B$

Teorema de terminación

Teorema

Si v es una expresión variante (con las propiedades de la p. 236) e I es un invariante (con las propiedades de la p. 234) de un ciclo y c es una cota tal que $(I \wedge v \leq c) \rightarrow \neg B$, entonces el ciclo termina.

Demostración.

- por el absurdo: supongamos que el ciclo no termina
- sea v_j el valor que toma v después de ejecutar el cuerpo del ciclo por j-ésima vez ($j \geq 0$)
- como v es estrictamente decreciente, $v_1 > v_2 > v_3 > \ldots$
- como v es de tipo Int, $v_j \in \mathbb{N}$ para todo j
- debe existir un k tal que $v_k \leq c$
- como vale $(I \wedge v \leq c) \rightarrow \neg B$, tenemos que para este k vale $\neg B$ y por lo tanto sale del ciclo (después de iterar k veces)
- llegamos a un absurdo porque habíamos supuesto que el ciclo no terminaba. Entonces el ciclo termina. □

¿Qué pasaría si v fuese de tipo Float? ¿Funciona igual la demo?

Observaciones sobre terminación

```
int dec1(int x) {
   while (x > 0)
      x = x - 1;
   return x;
}
```

```
int dec2(int x) {
   while (x != 0)
      x = x - 1;
   return x;
}
```

Supongamos que x no es negativo

- la implicación $(I \wedge v \leq c) \rightarrow \neg B$:
  - el invariante debería ser $I : x \geq 0$ en los dos casos
  - ambos devuelven 0
  - iteran la misma cantidad de veces (x veces)
    - ¿hace falta I en $(I \wedge v \leq c) \rightarrow \neg B$?
    - en los dos casos, $v = x$ y $c = 0$ deberían ser buenas
      $x \leq 0 \rightarrow x \not> 0$, luego no se usa I para *dec*1
      pero $x \leq 0 \not\rightarrow x == 0$, luego hay que usar I para *dec*2
      usando I en *dec*2 tenemos: $(x \geq 0 \wedge x \leq 0) \rightarrow x == 0$
- si c es una buena cota, cualquier cota $k < c$ también:
  - -2 es buena cota para *dec*1: $x \leq -2 \rightarrow x \leq 0 \rightarrow x \not> 0$
  - -2 es buena cota para *dec*2: $(x \geq 0 \wedge x \leq -2) \rightarrow$ cualquier cosa
    - ¡no hay trampa en este ejemplo!
  - si v es una buena función variante con cota k, $v' = v - k$ es una buena función variante con cota 0
    - entonces sin pérdida de generalidad, podemos suponer siempre una cota 0

Teorema de Correctitud

Teorema

Si un ciclo que termina satisface $P_C \rightarrow I$ y $(I \wedge \neg B) \rightarrow Q_C$ entonces el ciclo es correcto con respecto a su especificación.

Demostración.

Queremos ver que el ciclo es correcto para su especificación, es decir, queremos ver que para variables que satisfagan P_C, cuando el ciclo termina se satisface Q_C

- supongamos que las variables en el estado 1 satisfacen P_C
- como $P_C \rightarrow I$, entonces en el estado 1 se satisface I
- ejecutamos el ciclo (0 o más veces)
  - por definición de invariante (ver p. 234), en cada iteración, el invariante se restablece
- por hipótesis, el ciclo termina y en el estado 2 vale $\neg B$
- además, en el estado 2 vale I
- como $(I \wedge \neg B) \rightarrow Q_C$ entonces en el estado 2 vale Q_C

```
//estado 1
//vale P_C;
while (B) {
  //vale I ∧ B;
  cuerpo
  //vale I;
}
//estado 2
//vale Q_C;
```

□

Todo junto: Teorema del Invariante

Teorema

Sea `while (B) { cuerpo }` *un ciclo con guarda B, precondición P_C y poscondición Q_C. Sea I un predicado booleano, v una función variante y c una cota. Si valen*

1. $P_C \rightarrow I$
2. $(I \wedge \neg B) \rightarrow Q_C$
3. *el invariante se preserva en la ejecución del cuerpo, i.e. si $I \wedge B$ vale en el estado A entonces I vale en el estado B*

```
//estado A
cuerpo
//estado B
```

4. *v es decreciente, i.e. $v@A > v@B$*
5. $(I \wedge v \leq c) \rightarrow \neg B$

entonces para cualquier valor de las variables del programa que haga verdadera P_C, el ciclo termina y hace verdadera Q_C, i.e. el ciclo cumple con su especificación (P_C, Q_C).

Demostración.

Combinar el Teorema de Terminación (p. 237) y el Teorema de Correctitud (p. 239). □

Ejemplo de demostración de correctitud

Demostración de $P_C \rightarrow I$

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

1. $P_C \rightarrow I$:

   Supongamos que vale P_C y veamos que vale cada parte de I

   - $0 \leq i$: si $i == 0$ esto es trivial
   - $i \leq x$: si $i == 0$ y $x == 0$ esto es trivial
   - $s == \sum[0..i]$: como $s == 0$ y $i == 0$ hay que probar que $0 == \sum[0..0]$, pero esto es trivial

Ejemplo de demostración de correctitud

Demostración de $(I \wedge \neg B) \rightarrow Q_C$

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

2. $(I \wedge \neg B) \rightarrow Q_C$:

   Supongamos que vale $I \wedge \neg B$ y veamos que vale cada parte de Q_C por separado:

   - $i == x$: de I sabemos que $i \leq x$ y de $\neg B$ sabemos $i \geq x$
   - $s == \sum[0..x]$: ya vimos que $i == x$. De I sabemos $s == \sum[0..i]$.

Ejemplo de demostración de correctitud

Demostración de que el cuerpo preserva el invariante

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

3. el cuerpo preserva el invariante:

   Hacemos la transformación de estados del cuerpo:

```
//estado 1
//vale B ∧ I
//implica 0 ≤ i < x ∧ s == ∑[0..i]
  (justificación trivial)
      i = i + 1;
//estado 2
//vale i == 1 + i@1 ∧ s == s@1
      s = s + i;
//estado 3
//vale i == i@2 ∧ s == s@2 + i@2
```


Ejemplo de demostración de correctitud

Demostración de que el cuerpo preserva el invariante (cont.)

Transformación de estados del cuerpo

```
//estado 1
//vale B ∧ I
//implica 0 ≤ i < x ∧ s == ∑[0..i]
      i = i + 1;
//estado 2
//vale i == 1 + i@1 ∧ s == s@1
      s = s + i;
//estado 3
//vale i == i@2 ∧ s == s@2 + i@2
```

Queremos ver que vale I@3:

- $i@3 \geq 0$: sabemos $i@3 == i@2 == 1 + i@1$ y por el estado 1 sabemos $i@1 \geq 0$. Luego $1 + i@1 \geq 0$
- $i@3 \leq x@3$: ya vimos que $i@3 == 1 + i@1$. Como x no cambia porque es una variable de entrada que no aparece ni en *modifica* ni en *local*, $x@3 == x@1$. Del estado 1 sabemos $i@1 < x@1$. Sumando 1 en ambos lados, $1 + i@1 < 1 + x@1$. De lo que dijimos antes, $i@3 < 1 + x@3$ y esto es equivalente a $i@3 \leq x@3$.
- $s@3 == \sum[0..i@3]$: ya vimos que $i@3 == 1 + i@1$. Sabemos que $s@3 == s@2 + i@2 == 1 + s@1 + i@1$. Como $s@1 == \sum[0..i@1]$, entonces $s@3 == \sum[0..i@1] + 1 + i@1 == \sum[0..1 + i@1] == \sum[0..i@3]$.

Ejemplo de demostración de correctitud

Demostración de que v es decreciente

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

4. v es decreciente:
   Recordemos la transformación de estados

```
//estado 1
//vale B ∧ I
//implica 0 ≤ i < x ∧ s == ∑[0..i]
        i = i + 1;
//estado 2
//vale i == 1 + i@1 ∧ s == s@1
        s = s + i;
//estado 3
//vale i == i@2 ∧ s == s@2 + i@2
```

- $v@3 == x@3 - i@3 == \text{pre}(x) - i@3$
- $v@1 == x@1 - i@1 == \text{pre}(x) - i@1$
- ya vimos que $i@3 = 1 + i@1$
- luego $v@3 == \text{pre}(x) - (1 + i@1) < \text{pre}(x) - i@1 == v@1$

Ejemplo de demostración de correctitud

Demostración de $(I \wedge v \leq c) \rightarrow \neg B$

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $P : x \geq 0$
  asegura $r == \sum[0..x]$
$\}$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

5. $(I \wedge v \leq c) \rightarrow \neg B$:
   Supongamos que vale $I \wedge v \leq c$
   - $v \leq c$ es equivalente a $x - i \leq 0$, que es equivalente a $x \leq i$ y esto es $\neg B$

Programación imperativa

Clase 3

Algoritmos de búsqueda

Arreglos

- secuencias de una cantidad fija de variables del mismo tipo
- se declaran con un nombre, un tamaño y un tipo
  - `int a[10];`
  - arreglo de 10 elementos enteros
  - nos referimos a los elementos a través del nombre del arreglo y un índice entre corchetes
  - `a[0]`, `a[1]`,..., `a[9]`
- solamente hay valores en las posiciones válidas
  - de 0 a la dimensión menos uno
  - una referencia a una posición fuera del rango da error (en ejecución)
  - el tamaño se mantiene constante

Arreglos y listas

- ambos representan secuencias de elementos de un tipo
- los arreglos tienen longitud fija; las listas, no
- los elementos de un arreglo pueden accederse en forma independiente
  - los de la lista se acceden secuencialmente, empezando por la cabeza
  - para acceder al i-ésimo elemento de una lista, hay que obtener i veces la cola y luego la cabeza
  - para acceder al i-ésimo elemento de un arreglo, simplemente se usa el índice

Arreglos en C++

- los arreglos en C++ son referencias
  - pueden modificarse cuando son pasados como argumentos
- no hay forma de averiguar su tamaño una vez que fueron creados
  - el programa tiene que encargarse de almacenarlo de alguna forma
  - en la declaración de un parámetro no se indica su tamaño

problema ceroPorUno(a : [Int], tam : Int){
  requiere $tam == |a|$;
  modifica a;
  asegura $a == [\text{if } i == 0 \text{ then } 1 \text{ else } i \mid i \leftarrow \text{pre}(a)[0..tam)]$; }

```
void ceroPorUno(int a[], int tam) {
  int j = 0;
  while (j < tam) {
    // invariante 0 ≤ j ≤ tam ∧ a[j..tam) == pre(a)[j..tam)∧
    // a[0..j) == [if i == 0 then 1 else i | i ← pre(a)[0..j)];
    // variante tam − j;
    if (a[j] == 0) a[j] = 1;
    j++;
  }
}
```


Búsqueda lineal sobre arreglos

- Especificación

problema buscar (a : [Int], x : Int, n : Int) = res : Bool{
  requiere $n == |a| \land n > 0$;
  asegura $res == (x \in a)$;
}

- Implementación

```
bool buscar (int a[], int x, int n) {
     int i = 0;
     while (i < n && a[i] != x) {
          i = i + 1;
     }
     return i < n;
}
```

El algoritmo de búsqueda lineal

- es el más ingenuo
- revisa los elementos del arreglo en orden
- en el peor caso realiza n operaciones

Especificación del ciclo

```
bool buscar(int a[], int x, int n) {
   int i = 0;

// vale Pc : i == 0

   while (i < n && a[i] != x) {

// invariante I : 0 ≤ i ≤ n ∧ x ∉ a[0..i)
// variante v : n − i

      i = i + 1;
   }

// vale Qc : i < n ↔ x ∈ a[0..n)

   return i < n;
}
```

Correctitud del ciclo

```
// vale Pc
// implica I
   while (i < n && a[i] != x) {
// estado E
// vale I ∧ i < n ∧ a[i] ≠ x
       i = i+1;
// estado F
// vale I
// vale v < v@E
   }
// vale I ∧ ¬(i < n ∧ a[i] ≠ x)
// implica Qc
```

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow \neg(i < n \wedge a[i] \neq x)$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

Recordar que

- el invariante es $I : 0 \leq i \leq n \wedge x \notin a[0..i)$
- la guarda B es $i < n \wedge a[i] \neq x$

```
// estado E (invariante + guarda del ciclo)
// vale 0 ≤ i ≤ n ∧ x ∉ a[0..i) ∧ i < n ∧ a[i] ≠ x
// implica 0 ≤ i < n                    (juntando 0 ≤ i ≤ n y i < n)
// implica x ∉ a[0..i]                  (juntando x ∉ a[0..i) y a[i] ≠ x)
// implica x ∉ a[0..i + 1)              (propiedad de listas)

       i = i + 1;

// estado F
// vale i = i@E + 1
// implica 1 ≤ i@E + 1 < n + 1 (está en E y sumando 1 en cada término)
// implica 0 ≤ i ≤ n        (usando que i == i@E + 1 y propiedad de ℤ)
// implica x ∉ a[0..i@E + 1)            (está en E; a no puede cambiar)
// implica x ∉ a[0..i)                  (usando que i == i@E + 1)
// implica I                            (se reestablece el invariante)
```


2. La función variante decrece

```
// estado E (invariante + guarda del ciclo)
// vale I ∧ B

       i = i + 1;

// estado F
// vale i == i@E + 1
```

¿Cuánto vale $v = n - i$ en el estado F?

$$\begin{aligned} v@F &== (n - i)@F \\ &== n - i@F \\ &== n - (i@E + 1) \\ &== n - i@E - 1 \\ &< n - i@E \\ &== v@E \end{aligned}$$

5. La poscondición vale al final

Quiero probar que:

$$\overbrace{0 \leq i \leq n}^{1} \wedge \overbrace{x \notin a[0..i)}^{2} \wedge \overbrace{(i \geq n \vee x == a[i])}^{5}$$

$$\Downarrow$$

$$Qc : \underbrace{i < n}_{3} \leftrightarrow \underbrace{x \in a[0..n)}_{4}$$

Demostración:

- supongamos $i \geq n$ (i.e. 3 es falso). Por 1, $i == n$. Por 2, tenemos $x \notin a[0..n)$. Luego 4 también es falso.
- supongamos $i < n$ (i.e. 3 es verdadero). Por 5, $x == a[i]$. De 1 concluimos que 4 es verdadero.

3 y 4 son triviales

3. Si la función variante pasa la cota, el ciclo termina:

$$n - i \leq 0 \quad \Rightarrow \quad n \leq i \quad \Rightarrow \quad \neg B$$

4. La precondición del ciclo implica el invariante

$$i == 0 \quad \Rightarrow \quad 0 \leq i \leq n \wedge x \notin a[0..i)$$

Entonces el ciclo es correcto con respecto a su especificación. Observar que

// estado H
// vale $Qc : i < n \leftrightarrow x \in a[0..n)$

```
return i < n;
```

// vale $res == (i < n)@H \wedge i = i@H$
// implica $res == x \in a[0..n)$ (esta es la poscondición del problema)

Complejidad de un algoritmo

- máxima cantidad de operaciones que puede realizar el algoritmo
  - hay que buscar los datos de entrada que lo hacen trabajar más
- se define como función del tamaño de la entrada
  - $T(n)$, donde n es la cantidad de datos recibidos
- notación O grande
  - decimos que $T(n)$ es $O(f(n))$ cuando
    - $T(n)$ es a lo sumo una constante por $f(n)$
    - excepto para valores pequeños de n
  - $T(n)$ es $O(f(n))$ sii $(\exists c, n_0)(\forall n \geq n_0)\ T(n) \leq c \cdot f(n)$
  - se suele leer $T(n)$ es del orden de $f(n)$

Complejidad de la búsqueda lineal

- ¿cuándo se da la máxima cantidad de pasos?
  - cuando el elemento no está en la estructura
- ¿cuántos pasos son?
  - tantos como elementos tenga la estructura
- la complejidad del algoritmo es del orden del tamaño de la estructura
  - en un arreglo o lista, hay que hacer tantas comparaciones (y sumas) como elementos tenga el arreglo o lista
- si llamamos n a la longitud de la estructura
  - el algoritmo tiene complejidad $O(n)$
  - en cada paso tengo cuatro operaciones (dos comparaciones, una conjunción y el incremento del índice), entonces $c = 4$
    - pero esto no es importante
    - lo importante es cómo crece la complejidad cuando crece la estructura

Programación imperativa

Clase 4

Algoritmos de búsqueda

Mejorando la búsqueda

Supongamos que tenemos un diccionario y queremos buscar una palabra x.

¿Cómo podemos mejorar la búsqueda?

- abrimos el diccionario a la mitad
- si x está en esa página, terminamos
- si x es menor (según el orden de diccionario) que las palabras de la página actual, x no puede estar en la parte derecha
- si x es mayor (según el orden de diccionario) que las palabras de la página actual, x no puede estar en la parte izquierda
- seguimos buscando solo en la parte izquierda o derecha (según sea el caso)

Búsqueda binaria - especificación del problema

Supongamos que en lugar de un diccionario tenemos un arreglo ordenado y en lugar de buscar palabras buscamos números enteros.

Podemos hacer una búsqueda más eficiente

- revisamos solo algunas posiciones del arreglo: en cada paso descartamos la mitad del espacio de búsqueda
- menos comparaciones

Este tipo de algoritmo se llama búsqueda binaria.

Tenemos un nuevo problema en donde

- mantenemos la poscondición
- reforzamos la precondición

problema buscarBin (a : [Int], x : Int, n : Int) = res : Bool{
  requiere $|a| == n > 0$;
  requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;
  asegura $res == (x \in a)$;
}

El programa

```
bool buscarBin(int a[], int x, int n) {
   int i = 0, d = n - 1;
   bool res; int m;
   if (x < a[i] || x > a[d]) res = false;
   else {
      while (d > i + 1) {
         m = (i + d) / 2;
         if (x == a[m]) i = d = m;
         else if (x < a[m]) d = m;
         else i = m;
      }
      res = (a[i] == x || a[d] == x);
   }
   return res;
}
```


Especificación del ciclo

```
bool buscarBin(int a[], int x, int n) {
   int i = 0, d = n - 1; bool res; int m;
   if (x < a[i] || x > a[d]) res = false;
   else {
// vale Pc : i == 0 ∧ d == n − 1 ∧ a[i] ≤ x ≤ a[d]
      while (d > i + 1) {
// invariante I : 0 ≤ i ≤ d < n ∧ a[i] ≤ x ≤ a[d]
// variante v : d − i − 1
         m = (i + d) / 2;
         if (x == a[m]) i = d = m;
         else if (x < a[m]) d = m;
         else i = m;
      }
// vale Qc : 0 ≤ i < n ∧ 0 ≤ d < n ∧ x ∈ a ↔ (a[i] == x ∨ a[d] == x)
      res = (a[i] == x || a[d] == x);
   }
   return res;
}
```

Correctitud del ciclo

// vale Pc
// implica I

```
while (d > i + 1) {
```

// estado E
// vale $I \wedge d > i+1$

```
    m = (i + d) / 2;
    if (x == a[m]) i = d = m;
    else if (x < a[m]) d = m;
    else i = m;
```

// vale I
// vale $v < v@E$

```
}
```

// vale $0 \leq i < n \wedge 0 \leq d < n \wedge$
$x \in a \leftrightarrow (a[i] == x \vee a[d] == x)$
// implica Qc

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow d \leq i+1$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

// estado E (invariante + guarda del ciclo)
// vale $0 \leq i \wedge i+1 < d < n \wedge a[i] \leq x \leq a[d]$

```
m = (i + d) / 2;
```

// estado F
// vale $m = (i+d)@E \ div \ 2 \wedge i = i@E \wedge d = d@E$
// implica $0 \leq i < m < d < n$

```
if (x == a[m]) i = d = m;
else if (x < a[m]) d = m;
else i = m;
```

// vale $m == m@F$
// vale $(x == a[m@F] \wedge i == d == m@F) \vee$
$(x < a[m@F] \wedge d == m@F \wedge i == i@F) \vee$
$(x > a[m@F] \wedge i == m@F \wedge d == d@F)$

Falta ver que esto último implica el invariante

- $0 \leq i \leq d < n$: sale del estado F
- $a[i] \leq x \leq a[d]$:
  - caso $x == a[m]$: es trivial pues $a[i] == a[m] == a[d] == x$
  - caso $x < a[m]$: tenemos $a[i] \leq x < a[m] == a[d]$
  - caso $x > a[m]$: tenemos $a[i] == a[m] < x \leq a[d]$

2. La función variante decrece

// estado E (invariante + guarda del ciclo)
// vale $0 \leq i \wedge i+1 < d < n \wedge a[i] \leq x \leq a[d]$
// implica $d - i - 1 > 0$

```
m = (i + d) / 2;
```

// estado F
// vale $m == (i+d) \ div \ 2 \wedge i == i@E \wedge d == d@E$
// implica $0 \leq i < m < d < n$

```
if (x == a[m]) i = d = m;
else if (x < a[m]) d = m;
else i = m;
```

// vale $m == m@F$
// vale $(x == a[m@F] \wedge i == d == m@F) \vee$
$(x < a[m@F] \wedge d == m@F \wedge i == i@F) \vee$
$(x > a[m@F] \wedge i == m@F \wedge d == d@F)$

¿Cuánto vale $v = d - i - 1$?

- caso $x == a[m]$: es trivial pues $v = -1$
- caso $x < a[m]$: d decrece pero i queda igual
- caso $x > a[m]$: i crece pero d queda igual

3 y 4 son triviales

3. Si la función variante pasa la cota, el ciclo termina:

$$d - i - 1 \leq 0 \quad \Rightarrow \quad d \leq i + 1$$

4. La precondición del ciclo implica el invariante

$$Pc : i == 0 \wedge d == n - 1 \wedge a[i] \leq x \leq a[d]$$
$$\Downarrow$$
$$I : 0 \leq i \leq d < n \wedge a[i] \leq x \leq a[d]$$

5. La poscondición vale al final

Quiero probar que:

$$\overbrace{0 \le i \le d < n}^{1} \quad \wedge \quad \overbrace{a[i] \le x \le a[d]}^{2} \quad \wedge \quad \overbrace{d \le i + 1}^{3}$$

$$\Downarrow$$

$$\underbrace{0 \le i < n \wedge 0 \le d < n}_{4} \quad \wedge \quad \underbrace{x \in a}_{5} \leftrightarrow \underbrace{(a[i] == x \vee a[d] == x)}_{6}$$

Demostración:

- Por 1, resulta 4 verdadero.
- Supongamos 6 verdadero. De 1 concluimos que 5 es verdadero.
- Supongamos $a[i] \neq x \wedge a[d] \neq x$ (i.e. 6 es falso): $a[j] == x$ para algún j
  - $i < j < d$: contradice 3, $d \le i + 1$
  - $j < i$: gracias al orden, $x == a[j] < a[i]$; contradice 2, $x \ge a[i]$
  - $j > d$: gracias al orden, $x == a[j] > a[d]$; contradice 2, $x \le a[d]$

Entonces el ciclo es correcto con respecto a su especificación.

Complejidad

¿Cuántas comparaciones hacemos como máximo?

- en cada iteración me quedo con la mitad del espacio de búsqueda (tal vez uno más; no influye en el orden)
- paramos cuando el segmento de búsqueda tiene longitud 1 o 2

número de iteración	longitud del espacio de búsqueda
1	n
2	$n/2$
3	$(n/2)/2 = n/2^2$
4	$(n/2^2)/2 = n/2^3$
$\vdots$	$\vdots$
t	$n/2^{t-1}$

En total hacemos t iteraciones

$$1 = n/2^{t-1} \quad \Rightarrow \quad 2^{t-1} = n \quad \Rightarrow \quad t = 1 + \log_2 n.$$

Luego, la complejidad de la búsqueda binaria es $O(\log_2 n)$. Mucho mejor que la búsqueda lineal, que es $O(n)$.

Conclusiones

- vimos dos algoritmos de búsqueda
- en general, más información $\rightarrow$ algoritmos más eficientes

```
problema buscar (a : [Int], x : Int, n : Int) = res : Bool
    requiere |a| == n > 0;
    asegura res == (x ∈ a);

problema buscarBin (a : [Int], x : Int, n : Int) = res : Bool
    requiere |a| == n > 0;
    requiere (∀j ∈ [0..n − 1)) a[j] ≤ a[j + 1];
    asegura res == (x ∈ a);
```

- la búsqueda binaria es esencialmente mejor que la búsqueda lineal - $O(\log_2 n)$ vs. $O(n)$

Programación imperativa

Clase 5

Algoritmos de ordenamiento

Ordenamiento de un arreglo

- tenemos un arreglo de un tipo T con una relación de orden $(\leq)$
- queremos modificar el arreglo
  - para que sus elementos queden en orden creciente
  - vamos a hacerlo permutando elementos
- nuestra poscondición se va a parecer a la precondición de la búsqueda binaria

La especificación

problema sort<T> (a : [T], n : Int){
  requiere $1 \leq n == |a|$;
  modifica a;
  asegura mismos(a, pre(a)) $\wedge$ $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$;
}

aux cuenta(x : T, a : [T]) : Int = $|[y \mid y \leftarrow a,\ y == x]|$;

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

T tiene que ser un tipo con una relación de orden $\leq$ definida

El algoritmo Upsort

- ordenamos de derecha a izquierda
- el segmento desordenado va desde el principio hasta la posición que vamos a llamar *actual*
- comenzamos con $actual = n - 1$
- mientras $actual > 0$
  - encontrar el mayor elemento del segmento todavía no ordenado
  - intercambiarlo con el de la posición *actual*
  - decrementar *actual*

El programa

```
void upsort (int a[], int n) {
   int m, actual = n-1;
   while (actual > 0) {
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
}
```

problema maxPos(a : [Int], $desde$, $hasta$: Int) = pos : Int{
  requiere $0 \leq desde \leq hasta \leq |a| - 1$;
  asegura $desde \leq pos \leq hasta\ \wedge\ (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos}$;
}

problema swap(x, y : Int){
  modifica x, y;
  asegura $x ==$ pre(y) $\wedge$ $y ==$ pre(x);
}

Correctitud de Upsort

```
void upsort (int a[], int n) {
```

// vale $P : 1 \leq n == |a|$ (es la precondición del problema)

```
   int m, actual = n-1;
```

// vale $P_C : a == \text{pre}(a) \wedge actual == n-1$ (es la precondición del ciclo)

```
   while (actual > 0) {
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
```

// vale Q_C : mismos$(a, \text{pre}(a))$ $\wedge$ $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$ (es la poscondición del ciclo)

// vale Q : mismos$(a, \text{pre}(a))$ $\wedge$ $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$ (es la poscondición del problema)

```
}
```

Como $Q_C \Rightarrow Q$, lo que queda es probar que el ciclo es correcto para su especificación.

Especificación del ciclo

```
void upsort (int a[], int n) {
   int m, actual = n-1;
```

// vale $P_C : a == \text{pre}(a) \wedge actual == n-1$

```
   while (actual > 0) {
```

// invariante $I : 0 \leq actual \leq n-1 \wedge$ mismos$(a, \text{pre}(a))$
$\wedge\ (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
$\wedge\ actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$

// variante $v : actual$

```
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
```

// vale Q_C : mismos$(a, \text{pre}(a))$ $\wedge$ $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$

```
}
```


Correctitud del ciclo

// vale P_C
// implica I

```
   while (actual > 0) {
```

// estado E
// vale $I \wedge actual > 0$

```
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
```

// vale I
// vale $v < v@E$

```
   }
```

// vale $I \wedge \neg(actual > 0)$
// implica Q_C

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow \neg(actual > 0)$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

// estado E (invariante + guarda del ciclo)
// vale $0 < actual \leq n-1 \wedge$ mismos$(a, \text{pre}(a))$
$\wedge(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
$\wedge(actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1})$

```
         m = maxPos(a,0,actual);
```

// estado E_1

(Recordar especificación de MaxPos:
$P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1$, se cumple porque $0 < atual \leq n-1$
$Q_{MP} : desde \leq pos \leq hasta\ \wedge\ (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos}$)

// vale $0 \leq m \leq actual\ \wedge\ (\forall x \leftarrow a[0..actual])\ x \leq a_m$
// vale $a == a@E\ \wedge\ actual == actual@E$
// implica $0 < actual \leq n-1 \wedge$ mismos$(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
// implica $(actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1})$

(Justificación de los implica: *actual* y *a* no cambiaron
–lo dice el segundo vale de este estado)

1. El cuerpo del ciclo preserva el invariante (cont.)

// estado E_1
// vale $0 \leq m \leq actual \ \wedge \ (\forall x \leftarrow a[0..actual])\ x \leq a_m$
// implica $0 < actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
// implica $actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$

```
swap(a[m],a[actual]);
```

// estado E_2
// vale $a_m == (a@E_1)_{actual} \ \wedge \ a_{actual} == (a@E_1)_m$ (por poscon. de *swap*)
// vale $(\forall i \leftarrow [0..n), i \neq m, i \neq actual)\ a_i == (a@E_1)_i$ (idem)
// vale $actual == actual@E_1 \wedge m == m@E_1$
// implica $0 < actual \leq n-1$ (*actual* no se modificó)
// implica $\text{mismos}(a, \text{pre}(a))$ (el *swap* no agrega ni quita elementos)
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
($a(actual..n-1]$ no se modificó porque $m \leq actual$)
// implica $(\forall k \leftarrow (actual-1..n-1))\ a_k \leq a_{k+1}$

(del tercer vale de E_2: $m == m@E_1$ y $actual == actual@E_1$;
del primer vale de y último implica de E_1: $(a@E_1)_m \leq (a@E_1)_{actual+1}$;
del primer y segundo vale de E_2: $a_{actual} \leq a_{actual+1}$)

// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual}$ (del primer vale de E_1 y E_2)

1. El cuerpo del ciclo preserva el invariante (cont.)

// estado E_2
// implica $0 < actual \leq n-1$
// implica $\text{mismos}(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual-1..n-1))\ a_k \leq a_{k+1}$
// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual}$

```
actual--;
```

// estado E_3
// vale $actual == actual@E_2 - 1 \ \wedge \ a == a@E_2$
// implica $0 \leq actual@E_2 - 1 < n-1$ (de primer vale de E_3)
// implica $0 \leq actual \leq n-1$ (reemplazando $actual@E_2 - 1$ por $actual$)
// implica $\text{mismos}(a, \text{pre}(a))$ (de segundo implica de E_2 y $a == a@E_2$)
// implica $(\forall k \leftarrow (actual@E_2 - 1..n-1))\ a_k \leq a_{k+1}$ (por E_2)
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
(reemplazo $actual@E_2 - 1$ por $actual$)
// implica $(\forall x \leftarrow a[0..actual+1])\ x \leq a_{actual+1}$ (por E_2 + reemplazo)
// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
(por ser un selector más acotado)
// implica $actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
(pues q implica $p \rightarrow q$)

2 y 3 son triviales

2. La función variante decrece:

// estado E (invariante + guarda del ciclo)
// vale $I \wedge B$

```
m = maxPos(a,0,actual);
swap(a[m],a[actual]);
actual--;
```

// estado F
// vale $actual == actual@E - 1$
// implica $v@F == v@E - 1 < v@E$

3. Si la función variante pasa la cota, el ciclo termina:

$$actual \leq 0 \quad \text{es} \quad \neg B$$

4. La precondición del ciclo implica el invariante

Recordar que

- $P : 1 \leq n == |a|$
- $P_C : a == \text{pre}(a) \wedge actual == n-1$
- $I : 0 \leq actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a))$
$\wedge (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
$\wedge actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$

Demostración de que $P_C \Rightarrow I$:

- $1 \leq n \wedge actual == n-1 \quad \Rightarrow \quad 0 \leq actual \leq n-1$
- $a == \text{pre}(a) \quad \Rightarrow \quad \text{mismos}(a, \text{pre}(a))$
- $actual == n-1 \quad \Rightarrow \quad (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
porque el selector actúa sobre una lista vacía
- $actual == n-1 \quad \Rightarrow$
$actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
porque el antecedente es falso

5. La poscondición vale al final

Quiero probar que: $(\neg B \wedge I) \Rightarrow Q_C$

Recordemos

$$\neg B \wedge I \;:\; 0 \leq actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a)) \wedge (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1} \wedge actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual+1])\ x \leq a_{actual+1} \wedge actual \leq 0$$

$$Q_C : \text{mismos}(a, \text{pre}(a)) \ \wedge\ (\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$$

Veamos que vale cada parte de Q_C:

- mismos$(a, \text{pre}(a))$: trivial porque está en I
- $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$:
  - primero observar que $actual == 0$
  - si $n == 1$, no hay nada que probar porque $[0..n-1) == []$
  - si $n > 1$
    - sabemos $(\forall k \leftarrow (0..n-1))\ a_k \leq a_{k+1}$
    - sabemos que $(\forall x \leftarrow a[0..1])\ x \leq a_1$, entonces $a_0 \leq a_1$
    - concluimos $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$

Implementación de maxPos

problema maxPos $(a : [\text{Int}], desde, hasta : \text{Int}) = pos : \text{Int}\{$
  requiere $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1;$
  asegura $Q_{MP} : desde \leq pos \leq hasta \wedge (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos};$
$\}$

```
int maxPos(const int a[], int desde, int hasta) {
      int mp = desde, i = desde;
      while (i < hasta) {
            i++;
            if (a[i] > a[mp]) mp = i;
      }
      return mp;
}
```


Correctitud de maxPos

problema maxPos $(a : [\text{Int}], desde, hasta : \text{Int}) = pos : \text{Int}\{$
  requiere $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1;$
  asegura $Q_{MP} : desde \leq pos \leq hasta \wedge (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos};$
$\}$

```
int maxPos(const int a[], int desde, int hasta) {
//vale P_MP : 0 ≤ desde ≤ hasta ≤ |a| − 1    (precondición del problema)
      int mp = desde, i = desde;
//vale P_C : mp == i == desde                (precondición del ciclo)
      while (i < hasta) {
            i++;
            if (a[i] > a[mp]) mp = i;
      }
//vale Q_C : desde ≤ mp ≤ hasta ∧ (∀x ← a[desde..hasta]) x ≤ a_mp
                                             (poscondición del ciclo)
      return mp;
//vale Q_MP : desde ≤ pos ≤ hasta ∧ (∀x ← a[desde..hasta]) x ≤ a_pos
                                             (poscondición del problema)
}
```


Especificación del ciclo

```
// vale P_C : mp == i == desde

      while (i < hasta) {

// invariante I : desde ≤ mp ≤ i ≤ hasta ∧ (∀x ← a[desde..i]) x ≤ a_mp
// variante v : hasta − i

            i++;
            if (a[i] > a[mp]) mp = i;
      }

// vale Q_C : desde ≤ mp ≤ hasta ∧ (∀x ← a[desde..hasta]) x ≤ a_mp
```

Correctitud del ciclo

// vale P_C
// implica I

```
while (i < hasta) {
```

// estado E
// vale $I \land i < hasta$

```
    i++;
    if (a[i] > a[mp]) mp = i;
```

// vale I
// vale $v < v@E$

```
}
```

// vale $I \land i \geq hasta$
// implica Q_C

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow i \geq hasta$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

Recordar que *desde*, *hasta* y *a* no cambian porque son variables de entrada que no aparecen en *local* ni *modifica*

// estado E (invariante + guarda del ciclo)
// vale $desde \leq mp \leq i < hasta \land (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

```
i++;
```

// estado E_1
// vale $i == i@E + 1 \land mp = mp@E$
// implica $P_{if} : desde \leq mp \leq i \leq hasta \land (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$

(de E, reemplazando $i@E$ por $i - 1$ y cambiando el límite del selector adecuadamente)

```
if (a[i] > a[mp]) mp = i;
```

// vale $Q_{if} : desde \leq mp \leq i \leq hasta \land (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

(observar que en este punto, tratamos al `if` como una sola gran instrucción que convierte P_{if} en Q_{if}. La justificación de este paso es la transformación de estados de la hoja siguiente)

// implica I

(observar que I es igual que Q_{if}, pero en general alcanzaría con que Q_{if} implique I)

Especificación y correctitud del If

$P_{if} : desde \leq mp \leq i \leq hasta \land (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$
$Q_{if} : desde \leq mp \leq i \leq hasta \land (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

// estado E_{if}
// vale $desde \leq mp \leq i \leq hasta \land (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$

```
if (a[i] > a[mp]) mp = i;
```

// vale $(a_i > a_{mp@E_{if}} \land mp == i@E_{if} \land i == i@E_{if})$
$\quad \lor (a_i \leq a_{mp@E_{if}} \land mp == mp@E_{if} \land i == i@E_{if})$
// implica $desde \leq mp \leq i \leq hasta$

(operaciones lógicas; observar que *desde*, *hasta* y *a* no pueden modificarse porque son variables de entrada que no aparecen ni en *modifica* ni en *local*; observar que $i == i@E_{if}$)

// implica $a_{mp@E_{if}} \leq a_{mp} \land a_i \leq a_{mp}$ (Justificar...)
// implica $(\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$
// implica Q_{if}

2. La función variante decrece

// estado E (invariante + guarda del ciclo)
// vale $desde \leq i < hasta$

```
i++;
```

// estado E_1
// vale $i == i@E + 1$
// implica $desde \leq i \leq hasta$

```
if (a[i] > a[mp]) mp = i;
```

// estado F
// vale $i == i@E_1$
// implica $i == i@E + 1$

¿Cuánto vale $v = hasta - i$ en el estado F?

$$\begin{aligned} v@F &== (hasta - i)@F \\ &== hasta - i@F \\ &== hasta - (i@E + 1) \\ &== hasta - i@E - 1 \\ &< hasta - i@E \\ &== v@E \end{aligned}$$

3, 4 y 5 son triviales

3. Si la función variante pasa la cota, el ciclo termina:

$$hasta - i \leq 0 \quad \Rightarrow \quad hasta \leq i$$

4. La precondición del ciclo implica el invariante:
   Recordar que la precondición de maxPos dice

$$P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1$$

   y que *desde* y *hasta* no cambian de valor. Entonces

$$P_C : i == desde \wedge mp == desde$$
$$\Downarrow$$
$$I : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$$

5. La poscondición vale al final: es fácil ver que $I \wedge i \geq hasta$ implica Q_C

Entonces el ciclo es correcto con respecto a su especificación.

Complejidad de maxPos

¿Cuántas comparaciones hacemos como máximo?

- el ciclo itera $hasta - desde$ veces
- cada iteración hace
  - una comparación (en la guarda)
  - una asignación (el incremento)
  - otra comparación (guarda del condicional)
  - otra asignación (si se cumple esa guarda)
- antes del ciclo se hacen dos asignaciones
- todo estos detalles no importan para calcular el orden
- $O(\text{longitud del segmento}) = O(hasta - desde + 1)$
- peor caso: $desde == 0$ y $hasta == |a| - 1$
- la complejidad es $O(|a|)$ para el peor caso

Complejidad de Upsort

- el ciclo de Upsort itera n veces
  - empieza con $actual == n - 1$
  - termina con $actual == 0$
  - en cada iteración decrementa *actual* en uno
- una iteración hace
  - una búsqueda de maxPos
  - un swap
  - un decremento
- de estos pasos, el único que no es O(1), constante, es el primero
- ¿cuántos pasos hace maxPos en cada iteración?
  - siempre $O(hasta - desde + 1) = O(actual + 1)$
  - en la primera hace n (busca el máximo del segmento a ordenar)
  - en la segunda hace $n - 1$ (busca el segundo mayor)
  - en la tercera hace $n - 2$
  - y así (podríamos verlo por inducción) hasta 2
- el total de pasos es
  $O(n + (n-1) + \ldots + 2) = O(n * (n+1)/2 - 1) = O(n^2)$
- los mejores algoritmos de ordenamiento son $O(n \log n)$