Programación 2

Recursión

Recursión - Recursividad
Introducción

- Diccionario castellano
  recurrir
  - volver una cosa al sitio de donde salió; retornar, repetirse, reaparecer
    (poco frecuente)
  - recurrir a algo -> hacer uso de ello
    (más común)
- **Subprogramas recurrentes**
  - se invocan (llaman) a sí mismos
  - definidos en términos de sí mismos

Introducción (cont)

- ¿**Circularidad**?

  Recurrencia inútil:

  ```
  void P() { P(); }
  ```

  Termina con un error de ejecución: no hay más memoria (p.ej: “stack overflow”)

  ¿ Por qué ?

  - cada vez que un subprograma Q llama a otro R debe guardarse una indicación del punto en Q donde el control debe retornar al finalizar la ejecución de R
  - las llamadas a procedimientos pueden encadenarse arbitrariamente: Q1 $\rightarrow$ Q2 $\rightarrow$ Q3 $\rightarrow$ ... $\rightarrow$ Qn $\rightarrow$ ...

Introducción (cont)

Hay una estructura de datos donde se almacenan los sucesivos puntos de retorno. En general se tiene:

$$P \rightarrow Q_1 \rightarrow Q_2 \rightarrow Q_3 \rightarrow \ldots \rightarrow Q_n \rightarrow \ldots$$

donde Q_n es el que se está ejecutando. Paralelamente, se ha formado la estructura de "puntos de retorno"

$p_0, p_1, p_2, \ldots, p_{n-1}$

$p_0 \rightarrow$ punto de retorno en P
$p_1 \rightarrow$ punto de retorno en Q_1
...
$p_{n-1} \rightarrow$ punto de retorno en Q_{n-1}

Introducción (cont)

- La estructura crece con cada nueva llamada (un lugar) y decrece al terminar la ejecución de un subprograma
- La estructura se comporta como una PILA (análogo a una pila de platos)

p_{n-1}
p_{n-2}
…
…
p_1
p_0

- El tope de la pila es el punto donde debe retornarse el control tras la terminación del subprograma corriente.
- Por lo tanto, si el subprograma corriente llama a otro, el correspondiente punto de retorno debe colocarse como nuevo tope de la pila
- Y al finalizar un subprograma, se usa el tope como dirección de retorno y se lo remueve de la pila.

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	---	call Q3	---
p0 Cód-2	Cód-4		
---	---	---	---
		Cód-6	

STACK

p_0

Ejecución: Cód-1

Invocaciones

	P	Q1	Q2	Q3
	---	---	---	---
		Cód-3		
	---	---	---	---
	Cód-1		Cód-5	
	---	call Q2	---	---
				Cód-7
	call Q1	---	---	---
p1	---	---	call Q3	---
p0	Cód-2	Cód-4		
	---	---	---	---
			Cód-6	

STACK

p_1

p_0

Ejecución: Cód-1, Cód-3

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	p1 ---	call Q3	---
p0 Cód-2	Cód-4		
---	---	---	---
		Cód-6	
		p2	

Ejecución: Cód-1, Cód-3, Cód-5

STACK

p2

p_1

p_0

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	p1 ---	call Q3	---
p0 Cód-2	Cód-4		
---	---	---	---
		p2 Cód-6	

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7

STACK

p2

p_1

p_0

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
	p1		
---	---	call Q3	---
p0 Cód-2	Cód-4		
---	---	---	---
		Cód-6	
		p2	

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6

STACK

p_1

p_0

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	p1 ---	call Q3	---
Cód-2	Cód-4		
p0 ---	---	---	---
		Cód-6	

STACK

p_0

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	---	call Q3	---
p0 Cód-2	Cód-4		
---	---	---	---
		Cód-6	

STACK

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
	Cód-3		
---	---	---	---
Cód-1		Cód-5	
---	call Q2	---	---
			Cód-7
call Q1	---	---	---
---	---	call Q3	---
Cód-2	Cód-4		
---	---	---	---
		Cód-6	

STACK

vacío

Ejecución: **Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2**

Fin de la ejecución de P

Introducción (cont)

Volviendo al ejemplo de "recurrencia inútil":

void P() { P(); }

La pila se hace crecer infinitamente, pero (la memoria de) la máquina es finita, por lo tanto, en algún momento <u>no hay más memoria</u>

(stack overflow = desbordamiento de pila)

Introducción (cont)

- Ejemplo: (un poco) más útil

```
void P() {
        int x; cin >> x;
        if (EsPrimo(x)) … ; else … ;
        P();
}
```

  - En principio, permitiría implementar un programa interactivo
  - Pero también termina por desbordar el stack.

- Los anteriores son ejemplos de recurrencia infinita

  Estas recurrencias:

  - pueden tener sentido en principio (como en el segundo ejemplo)
  - pero terminan por desbordar la memoria (al menos con las implementaciones comunes de las llamadas a subprogramas)

Introducción (cont)

Otro ejemplo: **Factorial** (**n!** = n * (n-1) * (n-2) * … * 1)

```
int Fact (unsigned int n) {
        if (n>1) return n * Fact (n-1);
        else return 1;
}
```

- Cada ejecución de *Fact(m)* para *m* de tipo *int* es finita.
- En este ejemplo, para valores no demasiado grandes de n, Fact(n) puede ser demasiado grande (*int overflow*).
- Existirá un rango de valores de tipo *int* para los cuales la función anterior computa efectivamente los correspondientes factoriales.

Introducción (cont)

- Comparar con la versión iterativa:

```
int Fact (unsigned int n) {
    int f = 1;
    for (int i=2; i<=n; i++)  f = f * i;
    return f;
}
```

- La versión recurrente es más simple
  - Análoga a una definición matemática.
- La versión iterativa es más eficiente (no usa el stack)
  - Se acomoda mejor al esquema de máquinas de estados. En particular, podría darse que la versión recurrente terminara por desbordar la pila en casos en que la versión iterativa terminaría normalmente.

A ver si entendimos la introducción…

Procedimiento P (x)

  Si x = 0 entonces

    Imprimir x

  Sino

    Imprimir x

    P (x-1)

    Imprimir (-1) * x

El llamado P(3), ¿qué salida produce?

A ver si entendimos…

Procedimiento P (x)

Si x = 0 entonces

Imprimir x

Sino

Imprimir x

P (x-1)

Imprimir (-1) * x

p

STACK (p)

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2)

Se imprime: 3

A ver si entendimos…

Procedimiento P (x)

Si x = 0 entonces

Imprimir x

Sino

Imprimir x

P (x-1)

Imprimir (-1) * x

p

STACK (p)

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2) -> P(1)

Se imprime: 3, 2,

A ver si entendimos…

Procedimiento P (x)

Si x = 0 entonces

Imprimir x

Sino

Imprimir x

P (x-1)

Imprimir (-1) * x

p

STACK (p)

3) Imprimir (-1)*x, x=1

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2) -> P(1) -> P(0)

Se imprime: 3, 2, 1,

A ver si entendimos…

Procedimiento P (x)

  Si x = 0 entonces

    Imprimir x

  Sino

    Imprimir x

    P (x-1)

    Imprimir (-1) * x

p

STACK (p)

3) Imprimir (-1)*x, x=1

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2) -> P(1) -> P(0)

Se imprime: 3, 2, 1, 0

A ver si entendimos…

Procedimiento P (x)

Si x = 0 entonces

Imprimir x

Sino

Imprimir x

P (x-1)

Imprimir (-1) * x

p

STACK (p)

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Se imprime: 3, 2, 1, 0, -1,

A ver si entendimos…

Procedimiento P (x)

    Si x = 0 entonces

        Imprimir x

    Sino

        Imprimir x

        P (x-1)

        Imprimir (-1) * x

p

STACK (p)

1) Imprimir (-1)*x, x=3

Se imprime: 3, 2, 1, 0, -1, -2

A ver si entendimos…

Procedimiento P (x)

Si x = 0 entonces

Imprimir x

Sino

Imprimir x

P (x-1)

Imprimir (-1) * x

p

STACK (p)

Se imprime: 3, 2, 1, 0, -1, -2, -3   FIN

De recursión (de cola) a iteración

¿Cómo transformar este código a otro equivalente, sin recursión?

Procedimiento P (x)

Si CasoBase (x) entonces

AcciónBase (x)

Sino

AcciónAntes (x)

P (Transformación (x))

~~AcciónDespués (x)~~ **“recursión de cola”**

De recursión (de cola) a iteración

Procedimiento P' (x)

x' = x

Mientras NO CasoBase (x')

AcciónAntes (x')

x' = Transformación (x')

FinMientras

AcciónBase (x')

¿Es conveniente la recursión cuando es de cola?

De recursión a iteración

¿Cómo transformar este código a otro equivalente, sin recursión?

Procedimiento P(x)

Si CasoBase (x) entonces

AcciónBase (x)

Sino

AcciónAntes (x)

P (Transformación (x))

AcciónDespués (x)

De recursión a iteración

Procedimiento P' (x)

x' = x

Pila s Vacía

Mientras NO CasoBase (x')

AcciónAntes (x')

Aplilar (x', s)

x' = Transformación (x')

AcciónBase (x')

Mientras NO PilaVacía (s)

AcciónDespués (Tope (s))

DesapilarTope (s)

¿Es conveniente la iteración para una recursión que no es de cola?

Primeras conclusiones

- Usamos <u>subprogramas recurrentes</u>
  - **Operando** sobre **nuevos datos**
  - **Produciendo además otros efectos**

- Esto le da sentido a la circularidad
  Por ejemplo, podemos decir que la función *Fact* está definida en términos de sí misma.
  Esto sugiere una circularidad de la definición pero en realidad es una afirmación no demasiado precisa.

- En realidad, <u>para cada n</u>, <u>*Fact(n)* no está definido circularmente</u> (i.e. en términos de sí mismo) sino en términos de *Fact(n-1)* o bien (si n<=1) directamente (es decir: sin usar *Fact*).

Primeras Conclusiones (cont)

- El uso de recursión permite escribir programas cuyas computaciones son de largo variable.
- Solapamiento recurrencia / iteración. Redundancia:
  - Teóricamente, alcanza con una de las dos.
  - De hecho, pueden considerarse lenguajes
    - sin iteración
    - sin asignación
      (ver *Fact* recurrente, alcanza con el concepto de <u>función</u> que retorna un valor)
    - sin variables de estado

$\Rightarrow$ Esto es la base de los llamados

Lenguajes Declarativos

Primeras Conclusiones (cont)

Lenguajes Declarativos:

- Funcionales son particularmente interesantes
- Lógicos

Los lenguajes con variables de estado, asignación e iteración son llamados lenguajes **Imperativos**

- La mayoría de los lenguajes imperativos modernos admite recurrencia (Pascal, C, C++, entre otros).
- El uso de recurrencia permite desarrollar soluciones simples y elegantes. En muchos casos en que las correspondientes soluciones iterativas son demasiado complejas.
- También se da lo inverso.

Pero,
¿cómo programar recursivamente sobre distintos tipos de datos?

Esto es, ¿cómo programar recursivamente de manera correcta y en forma metodológica?

Orígenes

- Lógica: Teoría de los Números Naturales y de las Funciones Computables mecánicamente.
- En Matemática, los números naturales
  - usualmente se asumen como bien conocidos
  - se escriben en notación decimal
  - También en lenguajes como Pascal, C/C++, Java, ...
- En Lógica, los naturales se definen explícitamente.
- La idea es abstraerse de cualquier sistema de numeración posicional
  - Un sistema de numeración es de hecho un sistema de representación de números

Orígenes (cont)

- El sistema en base <u>b</u> usa <u>b</u> símbolos
  Ejemplo: dígitos

  dn dn-1 .... do

  de tal forma que el número representado es:

  dn * b^n + ... + do * b^0   (un polinomio)

- Tratamos de abstraernos de todas estas representaciones i.e. buscar una "más general" que podamos tomar como la definición (lo esencial) del concepto de número natural.
- Esto nos lleva a considerar el sistema de numeración más simple posible:

  <u>SISTEMA UNARIO</u>

Orígenes (cont)

- Sistema unario de numeración:
  - hay un sólo dígito : |
  - representamos los números como secuencias de ese dígito:

    || |||||
  - es conveniente tener una representación para el 0 (cero)
- Esto nos lleva a la definición de los números naturales
  Es un caso de ***definición Inductiva*** de un conjunto. Damos reglas para construir todos los elementos del conjunto:

  Regla 1 : **0** es un natural (N)

  Regla 2 : Si **n** es un natural entonces (**S n**) es otro natural

  Regla 3 : Esos son todos los naturales

Formalmente: conjuntos inductivos, pruebas por inducción y recursión

Regla 1 : **0** es un natural (N)

Regla 2 : Si **n** es un natural entonces (**S n**) es otro natural

Regla 3 : Esos son todos los naturales

- **0** y **S** son llamados (operadores) CONSTRUCTORES del conjunto N
- La Regla 3 permite justificar el
  - PRINCIPIO de DEMOSTRACIÓN por INDUCCIÓN ESTRUCTURAL
  - EL ESQUEMA DE RECURSIÓN ESTRUCTURAL:

Formalmente: conjuntos inductivos, pruebas por inducción y recursión

Regla 1 : **0** es un natural (N)

Regla 2 : Si **n** es un natural entonces (**S n**) es otro natural

Regla 3 : Esos son todos los naturales

PRINCIPIO de DEMOSTRACIÓN por INDUCCIÓN ESTRUCTURAL

Dada una propiedad P sobre naturales, si:

(1) P(**0**) vale (CB) y

(2) asumiendo P(**n**) (HI), demostrar P(**S n**) vale (TI).

entonces **P(n) vale para todo número natural n**

EL ESQUEMA DE RECURSIÓN ESTRUCTURAL:

***f* : *N* → …**

f(0) = …

f(S n) = … f(n)

Factorial

fact: N $\rightarrow$ N

fact(0) = 1
fact(S n) = (S n) * fact(n)

Ej: fact 3 = 3 * fact 2 = 3 * 2 * fact 1 = 3 * 2 * 1 * fact 0 = 3 * 2 * 1 * 1 = 6

```
int fact (unsigned int n) {
  if (n==0) return 1;
  else return n * fact(n-1);
}
```

Otro ejemplo

fact: N $\rightarrow$ N

fact(0) = 1
fact(S n) = (S n) * fact(n)

Sumatoria de los primeros naturales:

sum: N $\rightarrow$ N

sum(0)= 0
sum(S n) = (S n) + sum(n)

Más ejemplos

$+ : N \times N \rightarrow N$

$m + 0 = m$

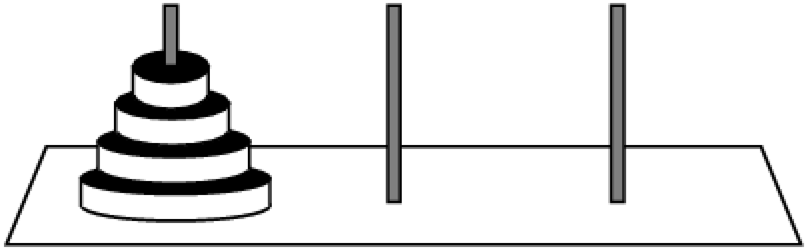
$m + (S\ n) = S\ (m + n)$

$* : N \times N \rightarrow N$

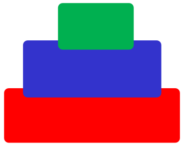
$m * 0 = 0$

$m * (S\ n) = (m * n) + m$

Torres de Hanoi

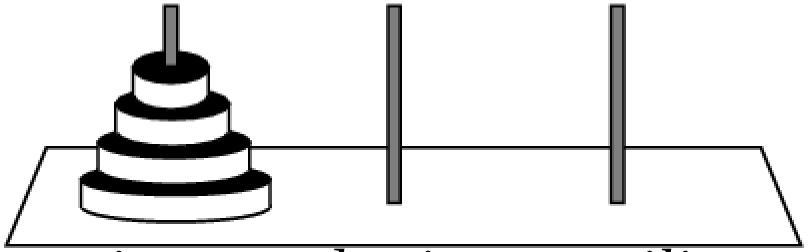


Origen *Destino* *Auxiliar*



Origen *Destino* *Auxiliar*

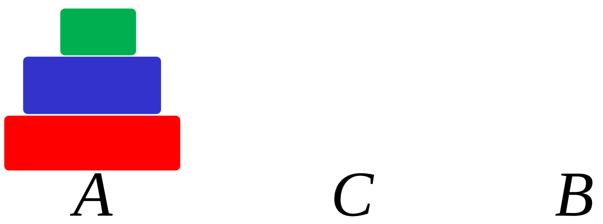
Hanoi




```
void hanoi(int n, char origen, char destino, char auxiliar){
      if(n > 0){

         /* Mover los n-1 discos de "origen" a "auxiliar" usando "destino" como auxiliar */
         hanoi(n-1, origen, auxiliar, destino);

         /* Mover disco n de "origen" para "destino" */
         printf("\n Mover disco %d de base %c para a base %c", n, origen, destino);

         /* Mover los n-1 discos de "auxiliar" a "destino" usando "origen" como auxiliar */
         hanoi(n-1, auxiliar, destino, origen);
}

main(){
         int n;
         printf("Digite el número de discos: ");
         scanf("%d",&n);
         hanoi(n, 'A', 'C', 'B');
         return 0;
}
```

Hanoi con n=3

```
void hanoi(int n, char origen, char destino, char auxiliar){
      if(n > 0){
          hanoi(n-1, origen, auxiliar, destino);
          printf("\n Mover disco %d de base %c para a base %c", n, origen, destino);
          hanoi(n-1, auxiliar, destino, origen);
}
```




```
hanoi(3, 'A', 'C', 'B')
          hanoi(2, 'A', 'B‘, 'C')
                    hanoi(1, 'A', ‘C‘, ‘B') = mover A -> C
                    mover A -> B
                    hanoi(1, ‘C', ‘B‘, ‘A') = mover C -> B
          mover A -> C
          hanoi(2, ‘B', ‘C‘, ‘A')
                    hanoi(1, ‘B', ‘A‘, ‘C') = mover B -> A
                    mover B -> C
                    hanoi(1, ‘A', ‘C‘, ‘B') = mover A -> C
```

Ahora Listas (Secuencias)

LISTAS:

- Vamos a definir el conjunto de las listas secuenciales finitas de naturales Inductivamente:
  - Regla 1**: lista vacía**

$$\frac{}{\mathbf{[]:Lista}}$$

  - Regla 2**: listas no vacías (cons)**

$$\frac{\mathbf{n:N} \qquad \mathbf{S:Lista}}{\mathbf{n.S:Lista}}$$

  - Regla 3**: esas son todas las listas**
- Ejemplos**:**

  []
  1.[] **([1])**
  3.1.[] **([3,1])** **notación sintética**

Recursión estructural en listas

f : Lista → …

f([]) = …
f(x.S) = … f(S)

Ejemplo:

largo : Lista → N

largo([]) = 0
largo(x.S) = 1 + largo(S)

Pertenece

– Chequear si un elemento está en una lista.

pertenece: N x Lista → bool

pertenece(e,[]) = …

pertenece(e,x.S) = … pertenece(e,S)

Pertenece

– Chequear si un elemento está en una lista.

pertenece: N x Lista → bool

pertenece(e,[]) = false

pertenece(e,x.S) = (e==x) || pertenece(e,S)

Otra (parecida):

cant: N x Lista → N

cant(e,[]) = 0

cant(e,x.S) = if (e==x) : 1 + cant(e,S)

else cant(e,S)

Inserción ordenada

– Insertar de manera ordenada un elemento en una lista ordenada.

Precondición: lista parámetro ordenada (≤)

insOrd: N x Lista → Lista

insOrd(e,[]) = …

insOrd(e,x.S) = … insOrd(e,S)

Ejemplo: insOrd(3,1.2.4.[]) = 1.insOrd(3,2.4.[]) =

1.2.insOrd(3,4.[]) = 1.2.3.4.[]

Inserción ordenada

– Insertar de manera ordenada un elemento en una lista ordenada.

Precondición: lista parámetro ordenada (≤)

insOrd: N x Lista → Lista

insOrd(e,[]) = e.[]

insOrd(e,x.S) = if (e<=x) : e.(x.S)

else x.insOrd(e,S)

Ejemplo: insOrd(3,1.2.4.[]) = 1.insOrd(3,2.4.[]) = 1.2.insOrd(3,4.[]) = 1.2.3.4.[]

Ordenación por inserción

– Ordenar una lista de menor a mayor.

Ord: Lista → Lista

Ord ([]) = …

Ord (x.S) = … Ord(S)

Ordenación por inserción

– Ordenar una lista de menor a mayor.

Ord: Lista $\rightarrow$ Lista

Ord ([]) = []

Ord (x.S) = insOrd(x,Ord(S))

Recursión General

Serie de Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, …

Fib: N $\rightarrow$ N (los números de Fibonacci)

Fib(0) = 1
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1), si n>=2

Fib(3) = Fib(1) + Fib(2) = 1 + Fib(0) + Fib(1) = 1 + 1 + 1 = 3

(dos llamadas en distintos puntos)

(no es un caso de recurrencia primitiva, estructural)

Exhaustividad + Exclusión + Terminación

Recursión General (cont)

- **Exhaustividad**: para cada n debe haber una ecuación (caso) que se aplique (para Fib: 0, 1 y n>=2 abarca todo N)
- **Exclusión**: Cada n tiene que corresponder a una única ecuación ó la función debe dar igual para los n que puedan entrar en más de una ecuación (para Fib: 0, 1 y los n>=2 son casos excluyentes).
- **Terminación**: Las llamadas recurrentes, para una función f, deben ser de la forma f(n) = c (f(m1), ... , f(mp)), donde mi < n para cada mi en un orden bien fundado (para Fib: n-1<n y n-2<n, siendo < un orden bien fundado para N).

⇒ **Esto da un criterio suficiente para garantizar la buena definición de una funcón f** (Fib es una función recursiva bien definida).

Se justifica por INDUCCIÓN COMPLETA (notar que < es "bien fundado")

Metodológicamente: pensar los casos de n

*** Base y * Reducción a un predecesor**

Principio de Inducción Completa

Si podemos probar P(n) asumiendo P(z) para todo z<n entonces vale **P(n) para todo n**

NOTAS:

- Así como la inducción primitiva (o estructural) se relaciona directamente con la recursión primitiva (o estructural), la inducción completa tiene su contraparte con la recursión general, que es equivalente a la Máquina de Turing.
- Si bien no es lo mismo probar que programar, pueden tratarse metodológicamente de la misma manera. Hay teorías y herramientas que se basan en esto para desarrollar sistemas correctos por construcción y verificar formalmente código. En particular esto se usa (y es muy útil y necesario) para sistemas críticos.

Algunas Conclusiones

- Los <u>números naturales</u>, las <u>listas</u>, los <u>árboles binarios</u>, entre otros, son conjuntos (tipos de datos) que pueden ser definidos **<u>inductivamente</u>** a través de reglas.

- Los conjuntos inductivos permiten:
  - » Probar propiedades por **<u>inducción primitiva</u> (<u>estructural</u>)**
  - » definir funciones por **<u>recursión primitiva</u> (<u>estructural</u>)**

- Es posible también definir **<u>funciones recursivas</u>** más generales, pero hay que probar existencia y unicidad. Tres condiciones suficientes para esto son:

Algunas Conclusiones (cont)

Las tres condiciones:

» **exhaustividad**

» **exclusión**

» **terminación: ver que cada llamado recursivo es más pequeño según un orden bien fundado**

(se justifica por inducción completa)

– Cuando una función recursiva tiene **precondición**, hay que asegurarse de que los parámetros con los que se llama a la función satisfagan la precondición.

» En particular, que los llamados recursivos de la función preserven el cumplimiento de su precondición.