Programación 2

Recurrencia
Inducción y 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 llamdas 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:

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

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

  $p_0, p_1, p_2, ..., p_{n-1}$

  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.

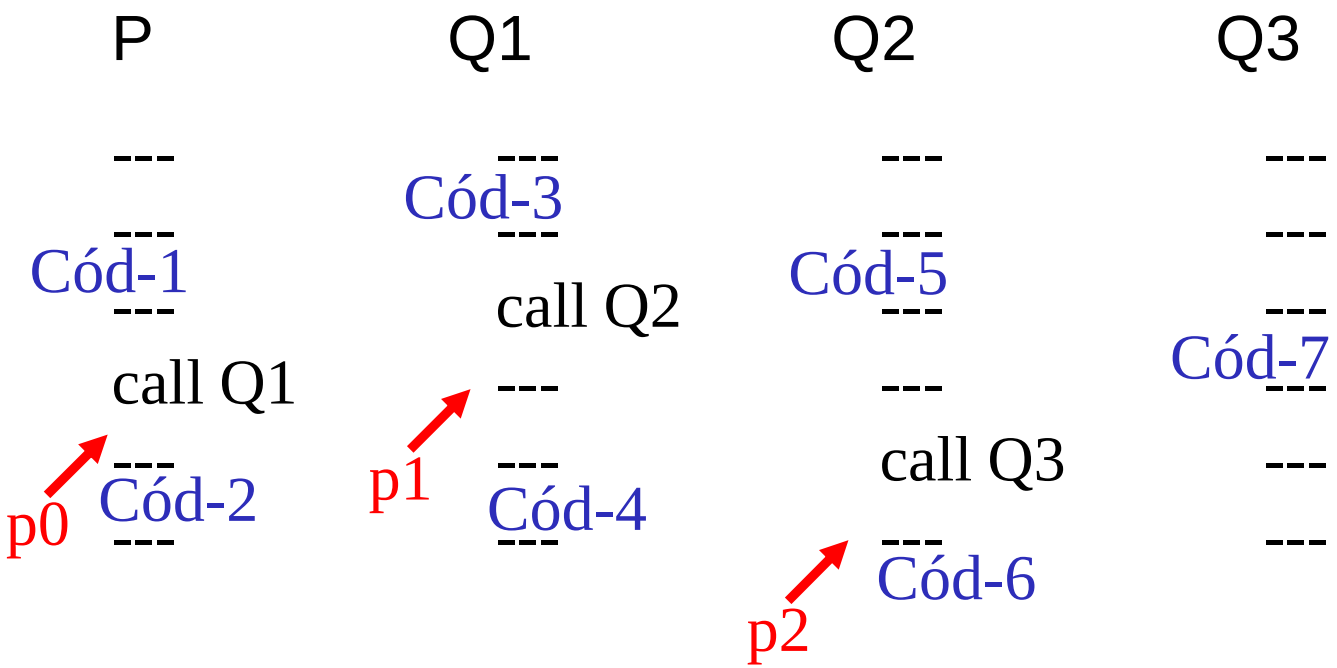
Introducción (cont)

PILA (LIFO) - Estructura lineal dinámica donde se agregan y quitan elementos sólo en uno de sus extremos.

COLA (FIFO) - Estructura lineal dinámica donde se agregan elementos en uno de sus extremos y se quitan del otro.

- Volviendo al ejemplo de "recurrencia inútil":
  la pila se hace crecer infinitamente, pero (la memoria de) la máquina es finita, por lo tanto, en algún momento no hay más memoria
  (stack overflow = desbordamiento de pila)

Introducción (cont)



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

STACK

p2

p_1

p_0

Introducción (cont)

- Ejemplo: (un poco) más útil

```
void P()
{ int x; cin >> x;
  if (EsPrimo(x))   cout << "Es Primo";   else cout << "No es Primo";
  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:

```
int Fact (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 versión iterativa:

```
int Fact (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…

```
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)

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, -1, -2, -3

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 subprogramas recurrentes
  - **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, para cada n, *Fact(n)* no está definido circularmente (i.e. en términos de sí mismo) sino en términos de *Fact(n-1)* o bien (si n==0) directamente (i.e sin usar *Fact*).

Primeras Conclusiones (cont)

- El cómputo de *Fact(n)* se realiza:
  - directamente (n = 0)
  - reduciéndolo a Fact de un número más chico (mas cercano a 0) **!!** Esto garantiza que toda ejecución de *Fact(n)* es finita y que por lo tanto *Fact(n)* esta bien definida para todo n (a menos del problema de ”Int overflow")

- El uso de recursión permite escribir programas cuyas computaciones son de largo variable.

Primeras Conclusiones (cont)

- 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 función 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.
- 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.

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 C/C++, Java, Pascal, ...)
- 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 b usa b símbolos
Ejemp.: 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:

SISTEMA UNARIO

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
  Regla 2 : Si n es un natural
  entonces (S n) es otro natural
  Regla 3 : Esos son todos los naturales

Un Conjunto Inductivo (nat) e Inducción

- Otra notación:
  - Regla 1: $\dfrac{}{\mathbf{0 : N}}$ Regla 2: $\dfrac{\mathbf{n : N}}{\mathbf{S\ n : N}}$

0 y **S** son llamados (operadores) CONSTRUCTORES del conjunto N

- La Regla 3 permite justificar el PRINCIPIO de DEMOSTRACIÓN por INDUCCIÓN MATEMÁTICA NATURAL (inducción primitiva)

  Sea P una propiedad de números naturales, o sea:

  P(n) es una proposición -enunciado (matemático)-

  Ejemplos

  n es par **n** > 2 **n** es primo

Principio de Inducción para nat

Entonces el siguiente es un principio (esquema) de demostraciones de enunciados de la forma:
P(n) vale para todo n.

(Obviamente, no un método para probar "P(n) vale para todo n" cualquiera sea P, sino para hacer evidente que "P(n) vale para todo n" para ciertas P)

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

(2) asumiendo que P(n) vale (HI) podemos demostrar que P(S n) vale (TI).

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

Principio de Inducción (cont.)

- **Idea**: "Juego de fichas de dominó":
  Para tirar todas las fichas:
  - tirar la primera
  - la distancia entre dos sucesivas debe ser tal que asegure que si cae la previa, entonces ella tira a la siguiente.

- **Otra idea**:
  La propiedada a probar debe ser:
  - una propiedad del padre (0)
  - ser hereditaria

Recursión Primitiva para nat

- La misma idea sirve para definir funciones sobre los naturales

  $f : N \rightarrow X$

- "tirar" -> asociarle su imagen en *f*, Definir *f(n)*

  f(0) = xo (no depende de f)
  f(S n) = c (n, f(n)) (donde c no depende de f)

  Si la función está definida en 0 y es hereditaria (podemos definirla en S n usando que está definida en n i.e. usando f(n)) entonces queda definida para todo n.
  (RECURRENCIA PRIMITIVA)

Recursión Primitiva (cont)

- Ejemplos:

fac: N $\rightarrow$ N

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

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

» Método mecánico de cálculo (simple sustitución)

» Otro modelo de cómputo (programa) mecánico (funciones recurrentes (recursivas)) (comparar con la máquina de estados)

+ : N x N $\rightarrow$ N

m + 0 = m
m + (S n) = S (m + n)

*** : N x N $\rightarrow$ N**

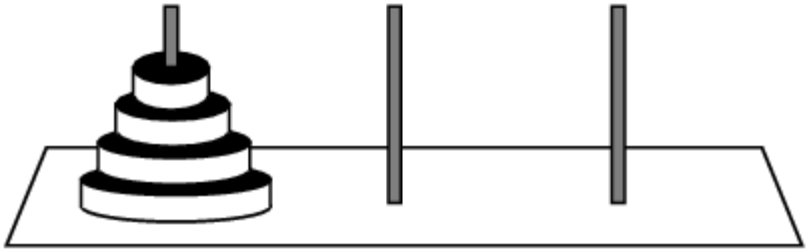
m * 0 = 0
m * (S n) = (m * n) + m

Recursión Primitiva (cont)

- En C++ usamos notación decimal en lugar de la unaria
  Las ecuaciones de la definición de fac se expresan:

```
if ( n==0 ) return 1;
   else return n * fac (n-1);
```

- Pensar el problema de las Torres de Hanoi



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;
}
```

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+2) = Fib(n) + Fib(n+1)
(dos llamadas en distintos puntos)

(no es un caso de recurrencia primitiva)

- **Exhaustividad**
- **Exclusión**
- **Terminación**

Principio de Inducción Completa

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

- En términos de las fichas de domino:

  Si una cualquiera se cae
  toda vez que todas sus predecesoras se caen
  entonces todas se caen

- Notar que toda aplicación de este principio requiere probar la propiedad para 0. (¿Por qué?)

Recursión General (cont)

Caso general de definición recurrente de funciones f : N $\rightarrow$ X

Casos Base (las bi no dependen de f)

f(n0) = b0

...

f(nk) = bk

Casos Recurrentes (las ei dependen de f)

f(nk+1) = e1

...

f(nk+r) = er

Para que *f* esté definida como función debe probarse,
para todo n: EXISTENCIA Y UNICIDAD de f(n)

Recursión General (cont)

- para cada n debe haber una ecuación (caso) que se aplique (**exhaustividad**)
- Cada n tiene que corresponder a una única ecuación (**exclusión**) ó f(n) debe ser igual para los n que violen exclus.
- (**terminación**) Las llamadas recurrentes deben ser de la forma f(n) = c (f(m1), ... , f(mp)) donde mi < n para cada mi en un orden bien fundado.

⇒ **Esto da un criterio suficiente para garantizar la buena definición de f.**

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

- **Metodológicamente:** pensar los casos de n
  *** Base y * Reducción a un predecesor**

Listas

LISTAS:

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

$$\frac{}{[] : \text{Nlista}}$$

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

$$\frac{n : N \qquad S : \text{Nlista}}{n.S : \text{Nlista}}$$

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

[]
1.[] ([1])
2.1.[] ([2,1]) notación sintética

Listas (cont)

- Tomamos en consecuencia:
  - **Principio de INDUCCIÓN PRIMITIVA ESTRUCTURAL (inducción primitiva):**

    Si P([]) y para todo n y S podemos probar P(n.S) asumiendo P(s) entonces P(S) vale para toda S : Nlista

    y tenemos también el esquema de definición de funciones sobre NLista por RECURRENCIA PRIMITIVA ESTRUCTURAL (recurrencia primitiva):

    f : NLista → X

    f([]) = x0
    f(x.S) = c(x, S, f(S))

- También las listas se pueden poner en hilera, tal como las fichas de dominó...

Listas (cont)

- Todo lo anterior se generaliza trivialmente a listas de elementos de cualquier tipo
  - **ALista donde A es cualquier tipo (el tipo de los elementos de la lista)**
- **!!!** Notar que NLista es un conjunto infinito

  (comparar con tipos de vectores, que son conjuntos de secuencias de un largo dado)

- Ejemplos de funciones sobre listas definidas por recurrencia primitiva

largo : ALista → N

largo([]) = 0

largo(x.S) = 1 + largo(S)

snoc: A x Alista → Alista

snoc(x,[]) = [x]

snoc(x,y.S) = y.(snoc(x,S))

Listas (cont)

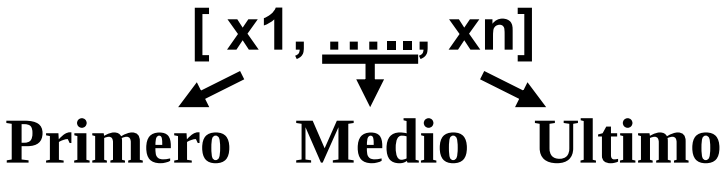
- **Descomposición de listas, más generalmente**
  - Problema: Escribir una función Pal : NLista → Bool tal que Pal(S) = true sii S es palíndroma capicúa

    Una manera de resolver el problema es:

    Pal([]) = true
    Pal([x]) = true
    Pal(S) = (Primero(S) = Ultimo(S)) & (Pal(Medio(S))
    (para S con al menos 2 elementos)

    donde las funciones Primero, Ultimo y Medio se definirán separadamente.

    Dada una lista con al menos dos elementos:



Listas (cont)

Por qué Pal es una función ?

– Se cumplen exhaustividad y exclusión.

– Se cumple la terminación:

! En la llamada recurrente, el argumento es más chico que en la llamada original

En particular, se da que para toda lista S de al menos dos elementos, largo(S) > largo(Medio(S))

El esquema general de definición de funciones recurrentes sobre listas es igual al visto para naturales

– En las llamadas recurrentes

f(S) = c (f(S1), ... , f(Sp))

Listas (cont)

- **Otros ejemplos**
  - **La función Medio no puede definirse para toda lista. De hecho:**

    Medio : {S : ALista / S tiene al menos dos elementos} $\rightarrow$ ALista

    Podemos decir: Medio : ALista $\rightarrow$ ALista
    con la precondición: el argumento debe tener al menos dos elementos

  - **Ejemplos similares puenden darse con naturales:**
    mcd : N x N $\rightarrow$ N (máximo común divisor)

    Precondición : los argumentos no pueden ser ambos nulos

Listas (cont)

- La función **Medio**

  Medio([x,y]) = []

  Medio(x.y.S) = y.Medio(y.S) (S no vacía)

  – **Notar los casos considerados. El espacio (dominio) son las listas de al menos dos elementos.**
  Se cumplen exhaustividad y exclusión (también terminación)

  – **En el caso de recurrencia, la llamada recurrente respeta la precondición de la función.**
  Fundamental!!!: si se usa una función sin respetar la precondición, no se puede tener ninguna garantía acerca del resultado

  – **Ver que la definición es correcta: S = [z1, ... , zn, zn+1] (n≥0)**

  Medio(x.y.S) = Medio ([x,y,z1, ... , zn,zn+1]) = [y, z1, ... , zn] = y.Medio(y.S), tal como está definida. (Nota: Medio(y.S) = [z1, ... , zn])

Algunas Conclusiones

- Los números naturales, las listas, los árboles binarios, entre otros, son conjuntos que pueden ser definidos **inductivamente** a través de reglas.

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

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

Algunas Conclusiones (cont)

Las 3 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.

Ejercicios
Resolver en la clase

- Chequear si un elemento está en una lista.
- Eliminar la primera ocurrencia de un elemento de una lista.
- Eliminar todas las ocurrencias de un elemento de una lista.
- Insertar de manera ordenada un elemento en una lista ordenanda.
- Ordenar una lista usando la función previa (*insert sort*).
- Eliminar duplicados de una lista, dejando solo una ocurrencia (la primera) de cada elemento diferente.

Ejercicios
Resolver en la clase

– Chequear si un elemento está en una lista.

pertenece: A x ALista → bool

pertenece (e, []) = false

pertenece (e, x.S) = true, Si e=x

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

Ejercicios
Resolver en la clase

– Eliminar la primera ocurrencia de un elemento de una lista.

elim: A x ALista → ALista

elim (e, []) = []

elim (e, x.S) = S, Si e=x,

elim (e, x.S) = x.elim(e,S), Sino

Ejercicios
Resolver en la clase

– Eliminar todas las ocurrencias de un elemento de una lista.

Adaptar la función **elim** previa:

elimT: A x ALista → AListaelimT (e, []) = []

elimT (e, x.S) = elimT(e,S), Si e=x

elimT (e, x.S) = x.elimT(e,S), Sino

Ejercicios
Resolver en la clase

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

insOrd: A x ALista → ALista

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

insOrd (e, x.S) = e.x.S, Si e<=x

insOrd (e, x.S) = x.insOrd(e,S), Sino

Ejercicios
Resolver en la clase

– Ordenar una lista usando la función previa (*insert sort*).

Ord: ALista → ALista

Ord ([]) = []

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

Ejercicios
Resolver en la clase

– Eliminar duplicados de una lista, dejando solo una ocurrencia (la primera) de cada elemento diferente.

ElimRep: ALista → ALista

ElimRep ([]) = []

ElimRep (x.S) = x.ElimRep(elimT(x,S))

Ejemplo: [a, b, f, b, g, c, a, b, f, w, a]

Resultado: [a, b, f, g, c, w]

Analizar porqué ElimRep es función.

Ejercicios adicionales sobre naturales

En los 5 ejercicios siguientes justifique que la definición dada corresponde a una función.

1) Considere la función f : N->N

   f (0) = 0 y f (S n) = (S n) + f (n)

   Pruebe que para todo n:N se cumple f (n) = n*(n+1)/2

2) Considere la función g : N->N

   g (0) = 1 y g (S n) = g (n) + g (n)

   Pruebe que para todo n:N se cumple g (n) = 2^n

3) Considere la función h : N×N->Z

   h (n, 0) = n y h (n, S m) = h (n,m) - 1

   Pruebe que para todo n,m:N se cumple h (n, m) = n-m

Más ejercicios sobre naturales

4) Defina una función *resta*: N×N->N que calcule la resta *natural* de dos números naturales.

5) Defina una función *div*: N×N->N que calcule la división entera de dos números por recursión en el primer argumento usando restas sucesivas (notar que la función tiene una precondición).

6) Implemente las funciones anteriores en C++.

Ejercicios propuestos sobre listas

En los ejercicios siguientes justifique que la definición recursiva dada corresponde a una función.

1) Defina una función que dada una lista *L* y un elemento *e,* cuente la cantidad de veces que ocurre *e* en *L*.

2) Defina una función que dada una lista *L* retorne 1 si la lista tiene longitud par y 0 sino (sin usar funciones auxiliares).

3) Defina una función que concatene (++) dos listas, esto es, dadas $L_1=[a_1,\ldots,a_n]$ y $L_2=[b_1,\ldots,b_m]$, devuelva la lista $[a_1,\ldots,a_n, b_1,\ldots,b_m]$.

4) Defina una función inv que invierta una lista, esto es, dada $L=[a_1,\ldots,a_n]$, devuelva la lista $[a_n,\ldots,a_1]$.

Más ejercicios sobre listas

5) Defina una función que retorne el mínimo elemento de una lista de números naturales (tiene precondición?).

6) Defina una función que dada una lista de elementos del conjunto {0, …, 9}, retorne el número natural que la lista representa. Ej: [2,4,7] => 7 + 10*4 + 100*2.

7) Demuestre las siguientes propiedades sobre listas arbitrarias L1 y L2:

(P1) L1++[] = L1

(P2) asociatividad de la concatenación (++),

(P3) inv(L1++L2) = inv(L2)++inv(L1)

(P4) inv(inv(L1)) = L1

Algunas Conclusiones

- Los números naturales, las listas, los árboles binarios, entre otros, son conjuntos que pueden ser definidos **inductivamente** a través de reglas.

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

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

Algunas Conclusiones (cont)

Las 3 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.