

# Aplicaciones de la Recursividad a la Programación

# La Programación como Composición

- Un programa consta de *sentencias ejecutables*, *declaraciones*, *comentarios*, etc.
- Las *sentencias ejecutables* alteran el valor de variables – **El estado del programa.**
- Una *sentencia ejecutable* es una función: Toma el estado inicial y lo transforma en un estado final (si se excluyen loops sin fin y excepciones).

# Sentencias como composiciones

- Llamemos  $x$  al estado inicial del programa ( $x$  representa los valores de todas las variables en el programa).
- La ejecución de una primera sentencia modifica el estado produciendo  $A(x)$ .
- La ejecución de una segunda sentencia  $B$ , toma como entrada el estado  $y$ , resultado de la ejecución de  $A$ , por lo tanto obtiene  $B(A(x))$
- También se representa

$A; B$

$B \circ A$

# Ejemplos

- Sean las sentencias:

$a = a+3; a=a*2;$

- Calcular
  - ♦  $(a=a+3)(x)$
  - ♦  $(a=a*2)(y)$
  - ♦  $(a=a+3; a=a*2)(x)$

# Notas sobre la composición

- Si se tiene la secuencia de sentencias ejecutables A; B; C;, el resultado es  $C(B(A(x)))$
- Observemos que agrupaciones de sentencias, tales como (A; B); C; ó A; (B; C); producen el mismo resultado.

***La composición de funciones se comporta como un operador asociativo***

No son necesarios los paréntesis

$$C \circ B \circ A = (C \circ B) \circ A = C \circ (B \circ A)$$

# La función identidad

- No altera el estado del programa
- Se representa

$$I(x) = x$$

# Sentencias condicionales como funciones

La sentencia

```
if g(x) then A;
```

```
else B;
```

se representa mediante la función

$$H = ( \text{ if } g(x) \text{ then } A \text{ else } B )$$

# Ciclos como funciones recursivas

El ciclo

```
while g(x) A;
```

Se puede representar como la función

$$W = (\text{while } g(x) \text{ } A)$$

o haciendo explícita la recursión

$$W(x) = (\text{if } g(x) \text{ then } (A; W)(x))$$

***En general todo programa puede traducirse como un composición de un conjunto de funciones primitivas***<sub>8</sub>



# Recursividad en Programación

- Similar a las definiciones recursivas, los procedimientos recursivos se invocan a si mismos, salvo para los casos base.
- Un procedimiento recursivo debe resolver uno o más casos base de forma no recursiva y en los demás casos hacer uso de la llamada recursiva.
- La técnica de demostración por recursividad nos permite probar el correcto funcionamiento de estos procedimientos.

# Aserciones

- Son funciones del estado que se evalúan como verdaderas o falsas.
- Se utilizan en la definición de especificaciones de software:
  - ♦ Pre-condiciones
  - ♦ Post-condiciones
- Ejemplo: en java se tiene  
**`assert(<una expresión booleana>);`**

# Invariantes

- Predicados en el estado del programa que deben ser verdaderos para todos los estados válidos de su ejecución.

# Pre-condiciones y Post-condiciones

- Una aserción sobre el estado inicial de un segmento de código se llama una **pre-condición**.
- Una aserción sobre el estado final de un segmento de código se llama una **post-condición**.

# Especificaciones en Ing. de Software

- Las pre-condiciones establecen las condiciones que se deben cumplir antes del inicio del programa (o función o método).
- Las post-condiciones establecen las condiciones que deben cumplirse una vez concluye el programa.
- El diseño e implementación del programa consisten en plantear y codificar un algoritmo que dadas las pre-condiciones, siempre satisfaga las post-condiciones.

# Corrección parcial y total

- Un segmento de código es ***parcialmente correcto*** con respecto a la pre-condición  $P$  y la post-condición  $Q$ , si el estado final siempre satisface  $Q$ , siempre y cuando empiece en  $P$  y concluya.
- Un segmento de código es ***totalmente correcto*** si siempre que comienza en  $P$  termina y el estado final satisface  $Q$ .

# Ejemplo: Función factorial

- Si se definen:
  - Pre-condición:  $n$  es natural
  - Post-condición: el valor retornado es  $n!$
  - La función factorial es totalmente correcta.
- Si se definen:
  - Pre-condición:  $n$  es un entero
  - Post-condición: el valor retornado es  $n!$
  - La función factorial es parcialmente correcta.
- Implementación:
  - La misma función factorial implementada en Java falla para  $n$  grande.

# Demostración de corrección por recursividad

- Si se demuestran los casos base y el paso inductivo, el segmento de código es parcialmente correcto.
- Si además se demuestra que el dominio es bien fundado, el programa es totalmente correcto.



# Ejemplo: Búsqueda binaria

Dados:

- Un vector ordenado ascendentemente,
- un intervalo definidos por las posiciones primero-último donde se hace la búsqueda,
- una clave a buscar

Devolver

- encontrado = V/F si la clave fue encontrada en el vector
- posición, el índice del vector donde fue encontrada la clave.

# Procedimiento recursivo

- Si primero > último, entonces encontrado=falso.
- El arreglo se parte en 2 mitades a partir del punto:

$$\text{medio} = (\text{primero} + \text{ultimo}) \div 2$$

- si  $\text{vector}[\text{medio}] < \text{clave}$ , buscar en el subintervalo derecho,
- si  $\text{vector}[\text{medio}] > \text{clave}$ , buscar en el subintervalo izquierdo,
- de lo contrario, el dato o está en medio o no está presente.

# Pseudo-código

```
funcion busquedaBinaria(primer, ultimo, clave, vector)
if primer>ultimo,
    devolver falso
else
    medio ← (primer+ultimo)÷2
    if clave<vector[medio]
        devolver buscar(primer, medio-1, clave, vector)
    else if clave>vector[medio]
        devolver buscar(medio+1, ultimo, clave, vector)
    else
        posicion ← medio
        encontrado ← verdadero
        devolver posicion
```

# Análisis de corrección (I)

- **Base inductiva:** El intervalo vacío, correspondiente a  $\text{primero} > \text{ultimo}$ . Retorna falso, lo cual siempre es correcto.
- **Hipótesis inductiva:** Dado un intervalo  $[\text{primero}, \text{ultimo}]$ , suponemos que el procedimiento siempre es correcto para cualquier sub-intervalo.
- **Paso inductivo:** 3 casos
  - $\text{clave} < \text{vector}[\text{medio}]$ : Correcto por hipótesis inductiva
  - $\text{clave} > \text{vector}[\text{medio}]$ : Correcto por hipótesis inductiva
  - $\text{clave} = \text{vector}[\text{medio}]$ : El programa devuelve  $\text{posición} = \text{medio}$  si el dato está lo cual es correcto; o devuelve que no el dato no está.

# Análisis de corrección (II)

- **Dominio bien fundado:** El dominio es bien fundado porque en todas las secuencias de ejecución el intervalo siguiente es un sub-intervalo propio del elemento precedente.  
(Alternativamente: Dar una definición recursiva de intervalos propios de un vector ordenado)
- **Conclusión:** El procedimiento funciona correctamente para todos los casos que cumplen la pre-condición (vector ordenado ascendentemente).