4. El Pseudolenguaje y Construcción de programas correctos

Un programa en el pseudolenguaje tendrá la misma forma de una especificación:

[Declaración de variables
 { Precondición }
 Programa
 { Postcondición }
].

Ahora pasaremos a describir las acciones elementales de nuestro pseudolenguaje, así como los *constructores* del pseudolenguaje, que también llamaremos *instrucciones* del pseudolenguaje, y que determinan el control del flujo de ejecución de un programa. Por cada constructor S daremos además , una regla que permite probar formalmente que S cumple una especificación dada. Estas reglas se deducirán de la interpretación operacional de cada instrucción y la interpretación operacional de la proposición {P} S {Q} (recordemos: cada ejecución de S termina en un estado que cumple Q cuando es aplicada a un estado que cumple P).

4.1. Acciones elementales y tipos no estructurados en el pseudolenguaje

Los programas actúan sobre objetos. En el cambio de estado de estos objetos es donde se refleja la acción que el programa describe. Los objetos serán referenciados por variables. Los nombres de las variables las colocaremos en minúsculas (a menos que sean constantes) comenzando por una letra.

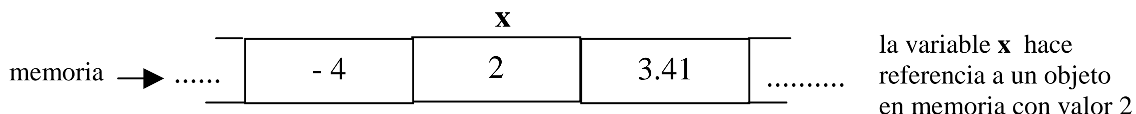

Figura 4

En un computador, un objeto será almacenado en la memoria y la variable que lo referencia la podemos ver como una dirección de memoria donde se encuentra el objeto almacenado. La memoria del computador la podemos imaginar como un casillero donde se almacenan los valores de las variables de nuestro programa. Cada casilla será un objeto diferente (ver figura 4).

Las acciones básicas son las de *observación* del valor de un objeto y *modificación* del valor de los objetos.

La observación de un valor de un objeto se describirá por el simple uso de su nombre. Por ejemplo, al evaluar la expresión (x+y)*2 en la cual aparecen dos variables de tipo entero,

x representa el valor del objeto x, e y representa el valor del objeto y; por lo tanto se observa el valor de x y de y.

4.1.1. Tipos no estructurados del pseudolenguaje

El tipo o clase de un objeto es la información necesaria y suficiente para poder conocer las manipulaciones que se podrán hacer con ese objeto. Por ejemplo, si x es un objeto de tipo entero, entonces podremos sumarlo, restarlo, multiplicarlo con otro objeto de tipo entero. Cuando decimos "un objeto es de tipo T" estamos implícitamente diciendo que el objeto puede poseer un valor de un conjunto preciso de valores y que sobre el valor de ese objeto podemos ejecutar determinadas operaciones.

En el pseudolenguaje podemos *declarar* variables de tipo entero, real, carácter, booleano. Las acciones elementales que podemos hacer con estos tipos de datos vienen dadas por las operaciones comúnmente utilizadas para éstos. De igual forma, las expresiones se forman por combinaciones válidas de los operadores, por ejemplo, las expresiones de tipo entero consisten de las constantes (representadas de manera usual, 2, 3, -4), variables de tipo entero, y combinaciones de estas formadas por operadores sobre enteros. Por ejemplo a*(b+2) es una expresión sobre números enteros que consta de dos operaciones elementales sobre los enteros.

Las siguientes operaciones elementales toman por operandos números enteros o reales:

+, -, *	suma, resta, multiplicación
/	división: produce un número real
$<, >, =, \geq, \leq, \neq$	operadores de comparación: producen verdadero o falso

Las siguientes operaciones tienen sólo enteros como operandos:

x div y	división entera de x entre y. Es la parte entera de x/y. No está definida para y=0
x mod y	resto de la división entera de x entre y. Es igual a x-(x div y)*y. Está definida sólo para x no negativo e y positivo

Las operaciones sobre tipos boléanos: $\neg, \wedge, \vee, \Rightarrow, \equiv$

Si a y b son enteros y p booleano entonces $(a \geq b-1) \wedge p$ es una expresión booleana o predicado.

Cualquiera de los tipos básicos admite como acción elemental la igualdad (=).

4.1.2. La Asignación

Cualquier cambio de estado que ocurra durante la ejecución de un programa, es debido a la modificación del valor de los objetos. Si a y b son dos variables que referencian a objetos del mismo tipo base, entonces podemos modificar el valor de a "asignándole" el

valor de b. Esta acción elemental la denotaremos en el pseudolenguaje por: a := b. más generalmente, una *asignación* es de la forma: x:=E, donde x es una variable y E es una expresión del mismo tipo que x, por ejemplo E=(b+c)*x.

La interpretación operacional de la acción de asignación es: la ejecución de x := E reemplaza el valor de x por el valor resultante de evaluar E.

En la figura 5 vemos el efecto que produce la asignación x := (b+c)*x para b = 3, c = 5 y x = 2.

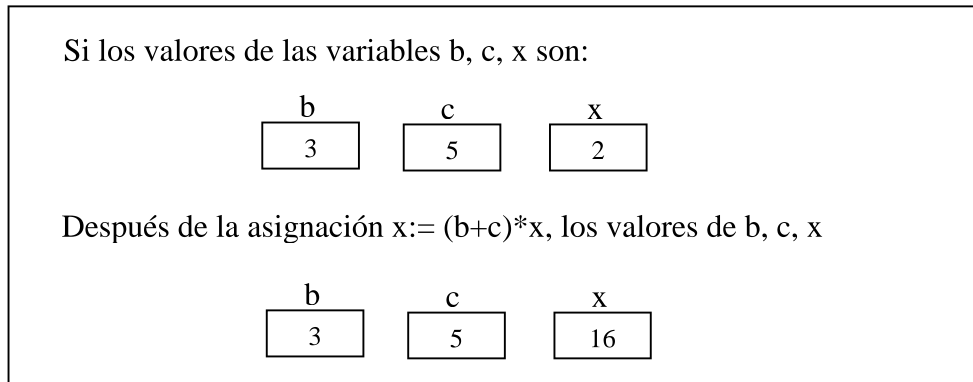

Figura 5

Si Q es un predicado que contiene la variable libre x, *Q(x:=E) es el predicado que resulta de sustituir cada aparición de la variable x en Q por la expresión E*. Dado un predicado Q, Q se cumple (es verdad) después de la ejecución de x := E si y sólo si Q(x:=E) se cumple antes de la ejecución de x:=E. Por lo tanto podemos definir formalmente *la regla que garantiza que una asignación cumple una especificación* de la siguiente manera:

Probar que {P} x:=E {Q} se cumple es equivalente a
probar que $P \Rightarrow Q(x:=E)$ es una tautología

Implícitamente se debe tener en cuenta que cuando P se cumple, este hecho debe implicar que la expresión E puede ser evaluada en el estado que describe P, es decir, E está bien definida para ese estado (por ejemplo, no hay división por cero).

Por ejemplo, mostremos que se cumple:

$$\{x \geq 3\}\ x := x+1 \{x \geq 0\}$$

Note primero (como observamos) que la expresión x+1 está bien definida para valores de x mayores o iguales que 3. Vemos que si la variable x es mayor o igual a 3 entonces al sumarle 1 será con más razón mayor o igual a cero. La demostración formal sería como sigue, partimos de la postcondición:

$(x \geq 0)(x:=x+1)$

≡ definición de la regla de sustitución (sustituir todas las apariciones de x por x+1)
    $x+1 \geq 0$
≡ por aritmética
    $x \geq -1$
⇐ por aritmética
    $x \geq 3$

Como $x \geq 3 \Rightarrow (x \geq 0)(x:=x+1)$ es una tautología, entonces la regla de la asignación nos dice que:

$$\{x \geq 3\} \quad x := x+1 \{x \geq 0\} \text{ se cumple}$$

Por lo tanto, siempre que $x \geq 3$ se cumpla (es decir, sea verdadero) después de ejecutar la instrucción $x := x+1$ se tendrá que $x \geq 0$ será verdadero. A los predicados $x \geq 3$, $x \geq 0$, también se les llama *aserciones* (o afirmaciones), en el sentido que estamos afirmando que son verdad en ese punto de la ejecución del programa.

¿Cúal sería la solución más débil X para que se cumpla: {X} x := E {Q}? Recordemos que X* es el predicado más débil que cumple {X} x := E {Q} si para todo predicado X que cumple {X} x := E {Q} se tiene que $X \Rightarrow X^*$. La solución es X = Q(x:=E), ya que para que Q se cumpla después de ejecutar la asignación, antes de ejecutar la asignación se debe cumplir Q reemplazando x por el valor por el que fue modificado, es decir, el valor de E. En el ejemplo anterior el predicado más débil es $x+1 \geq 0$.

Ejercicios: página 20 de Kaldewaij sección 0 de ejercicios.

4.1.3. La instrucción skip

Una instrucción que será necesaria más adelante es la instrucción **skip**. La ejecución de "salto" no tiene ningún efecto sobre el estado de las variables en el momento de su ejecución. La instrucción salto la caracterizamos por la regla:

Probar que {P} **skip** {Q} se cumple es equivalente a
Probar que $P \Rightarrow Q$ es una tautología

¿Cuál sería la precondición más débil de **skip**?. La respuesta es Q.

Ejercicios: página 17 del Kaldewaij.

4.2 Secuenciación de acciones

Cómo los programas que haremos describirán procesos secuenciales, necesitamos algún constructor que nos permita expresar una secuencia de instrucciones.

La secuenciación de las instrucciones S y T la denotamos por S**;**T.

La interpretación operacional es: primero se ejecuta S y luego T.

Para probar formalmente que se cumple {P} S;T {Q} tenemos que conseguir un predicado R tal que {P} S {R} y {R} T {Q} se cumplan. Por lo tanto tenemos *la regla de la secuenciación* siguiente:

Probar que{P} S;T {Q} se cumple es equivalente a
encontrar un predicado R tal que {P} S {R} y {R} T {Q} se cumplan

El punto y coma (;) no se usa como un terminador o separador sino como un operador de composición para combinar dos instrucciones.

La instrucción de secuenciación (el punto y coma) es una operación de composición de dos operaciones pues cuando queremos encontrar la precondición más débil X tal que {X} S;T {Q} se cumple, esta es la precondición X más débil que cumple {X} S {precondición más débil Y que cumple {Y} T {Q}}.

Ejemplos de desarrollo y prueba de programas:

1) Hacer un programa que cumpla la especificación siguiente:

[var v, k, l, x, r : reales;

{Sean V, K, L, X números reales no negativos. Y se tiene que v=V, k=K, l=L, x=X, con K y V distintos de cero, , donde L es la cantidad de litros consumidos por el vehículo cuando recorre K kilómetros a la velocidad constante V. }

GASOLINA CONSUMIDA

{ r es la cantidad de gasolina que consume el vehículo al recorrer X kilómetros a una velocidad constante V}

Queremos encontrar el programa GASOLINA CONSUMIDA. Recordemos que aplicando la técnica de diseño descendente, este problema lo descompusimos en dos problemas a resolver secuencialmente: primero el cálculo del número de litros de gasolina que consume por kilómetro el automóvil a velocidad constante V, y luego el cálculo de la gasolina consumida por el automóvil al recorrer X kilómetros a velocidad constante V.

El primer problema se resuelve mediante la expresión E = l/k
El segundo problema se resuelve mediante la expresión x*E, habiendo calculado E.

Sin embargo podemos colocar todo en una sola expresión r := x*(l/k). Y el programa quedaría:

[var v, k, l, x, r : reales;

{Sean V, K, L, X números reales no negativos, con K y V distintos de cero. Y se tiene que v=V, k=K, l=L, x=X.}

r := x*(l/k)

{ Q: r es la cantidad de gasolina que consume el vehículo al recorrer X kilómetros a una velocidad constante V. }

Note que la variable v no interviene en el programa, sólo sirve en la especificación, por lo que puede ser eliminada del programa. La demostración de la correctitud de este programa es muy sencilla: partimos de la postcondición Q y aplicamos la regla de la asignación:

Partiendo de la postcondición "Q: r es la cantidad de gasolina que consume el automóvil al recorrer X kilómetros a una velocidad constante V" y aplicando la regla de sustitución:

Q(r:=x*(l/k))

≡ regla de sustitución

x*(l/k) es la cantidad de gasolina que consume el automóvil al recorrer X kilómetros a una velocidad constante V, donde L es la cantidad de litros consumidos por el vehículo cuando recorre K kilómetros a la velocidad constante V.

⇐ por definición de igualdad

x*(l/k) = X*(L/K) ∧ X*(L/K) representa la cantidad de gasolina que consume un automóvil al recorrer X kilómetros a una velocidad constante V, donde L es la cantidad de litros consumidos por el vehículo cuando recorre K kilómetros a la velocidad constante V.

⇐ por definición de igualdad de x, l, k a X, L, K y resultados de la física

Sean V, K, L, X números reales no negativos, con K y V distintos de cero. Y se tiene que v=V, k=K, l=L, x=X.

Por lo tanto la regla de la asignación garantiza que el programa es correcto ya que:

(v=V, k=K, l=L, x=X, K y V distintos de cero) ⇒ Q(r:=x*(l/k))

2) Hacer un programa que cumpla la siguiente especificación:

[var x, y, z : entero;
{ x=X ∧ y=Y ∧ z=Z}

```
  S
  { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
].
```

De acuerdo a la postcondición, z depende de x modificado a 2*Z, x e y dependen sólo de los valores iniciales, por lo que primero habría que calcular x e y, luego z. Note también que y depende del valor inicial de x, por lo que si modificamos x primero que y perderíamos el valor inicial de x y no podríamos calcular y. En consecuencia, habría que calcular y antes que x. Notemos también que z depende del valor inicial de y. Pero si modificamos y para que contenga X+Y, perdemos el valor inicial de y. Sin embargo, este nuevo valor X+Y de y, aparece en el valor final de z, por lo que no importa si modificamos el valor inicial de y por X+Y, ya que este último valor es el que nos interesa para calcular z. Podríamos entonces reemplazar en la postcondición a z=x+2*(X+Y) por z=x+2*y.

Un posible programa sería:

```
[ var x, y, z : entero;
  { x=X ∧ y=Y ∧ z=Z}
    y := x + y;
    x := 2*z;
    z := x + 2 * y
  { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
].
```

Demostración de la correctitud del programa anterior:

Partiendo de la postcondición, iremos calculando aserciones intermedias (que se satisfacen entre dos instrucciones consecutivas del programa) de abajo hacia arriba hasta llegar a la precondición:

(x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y))(z := x + 2 * y)
≡ regla de sustitución
x=2*Z ∧ y= X+Y ∧ x + 2*y=x+2*(X+Y)
≡ simplificación usando aritmética y lógica
x=2*Z ∧ y= X+Y

Acabamos de probar que se cumple:

(1) { x=2*Z ∧ y= X+Y } z := x + 2 * y { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }

Así, antes de la instrucción z := x + 2 * y, se cumple x=2*Z ∧ y= X+Y. Por lo que la podemos tomar como postcondición de la instrucción x := 2*z :

(x=2*Z ∧ y= X+Y) (x := 2*z)
≡ sustitución x por 2*z

$2*z=2*Z \wedge y= X+Y$

Con esto último hemos probado que:

(2) { $2*z=2*Z \wedge y= X+Y$ } x := 2*z { $x=2*Z \wedge y= X+Y$ }

Por lo tanto (1) y (2) nos permiten concluir, por la regla de la secuenciación, que:

```
              { 2*z=2*Z ∧ y= X+Y }
                  x := 2*z ;
(3)               z := x + 2 * y
              { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
```

Ahora tomamos $2*z=2*Z \wedge y= X+Y$ como postcondición de la instrucción y := x + y :

$(2*z=2*Z \wedge y= X+Y)$ (y := x+y)
≡ sustitución y por x+y
$2*z=2*Z \wedge x+y = X+Y$
⇐ por igualdad
$x=X \wedge y=Y \wedge z=Z$

Lo anterior nos dice que se cumple lo siguiente:

(4) { $x=X \wedge y=Y \wedge z=Z$ } y := x + y { $2*z=2*Z \wedge y= X+Y$ }

(3) y (4) nos permiten concluir, aplicando la regla de la secuenciación, que:

```
{ x=X ∧ y=Y ∧ z=Z}
  y := x + y;
  x := 2*z;
  z := x + 2 * y
{ x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
```

El programa puede ser documentado con más detalle colocando entre cada par de instrucciones consecutivas los predicados que van resultando de la demostración de correctitud. Estos predicados son aserciones (o afirmaciones) en cada punto de la ejecución del programa, pues afirmamos que son verdad en ese punto de la ejecución del programa. Estas aserciones determinan un esquema de la demostración de la correctitud del programa. El programa documentado con todas sus aserciones sería:

```
[ var x, y, z : entero;
  { x=X ∧ y=Y ∧ z=Z}
  y := x + y;
  { 2*z=2*Z ∧ y = X+Y }
```

```
  x := 2*z;
  { x=2*Z ∧ y= X+Y }
  z := x + 2 * y
  { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
].
```

Un programa junto con una aserción entre cada par de instrucciones es llamado *“un esquema de demostración”* o *“un programa con anotaciones”* pues en efecto es un esquema de una demostración formal la cual puede ser reconstruida a cabalidad utilizando las reglas que caracterizan cada constructor del pseudolenguaje. Un programa con anotaciones es una manera de documentar detalladamente un programa, y permite a otra persona reconstruir fácilmente la demostración de correctitud del mismo.

Otro posible programa para la misma especificación anterior lo podríamos construir aumentando el espacio de estados con nuevas variables que contengan los valores iniciales de x, y, z. Esta solución es más costosa en términos de espacio, pues se requiere usar más casillas de memoria para almacenar los valores de las variables adicionales:

```
[ var x, y, z, x1, y1, z1 : entero;
  { x=X ∧ y=Y ∧ z=Z}
  x1 := x;
  y1 := y;
  z1 := z;
  x := 2*z1;
  y := x1 + y1;
  z := 2*z1 + 2*(x1 + y1)
  { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y) }
].
```

<u>Ejercicios</u>:

a) demostrar la correctitud del programa anterior.
b) Hacer varios programas (pudiendo aumentar el espacio de estados de ser requerido) que cumplan las especificaciones siguientes y demuestre la correctitud:

   [var x, y, z : entero;
     { x=X ∧ y=Y ∧ z=Z}
     S
     { x=2*Z ∧ y= X+Y ∧ z=x+2*(X+Y^2) }
   ].

   [var x, y: entero;
     { x=X ∧ y=Y }
     S
     { x=Y ∧ y=X}
   ].

c) Ejercicios página 22 del Kaldewaij.

Ejemplo de síntesis (espeficicación, desarrollo y prueba de programas):

Problema: Hacer un programa que dada una cantidad de segundos, convierta esa cantidad de segundos en días, horas, minutos y segundos. Un resultado del tipo: 3 días, 26 horas, 64 minutos, 90 segundos no es válido, ya que 90 segundos equivalen a 1 minuto y 30 segundos y hemos podido agregar ese minuto al número de minutos resultando así 65 minutos. Pero de nuevo, 65 minutos equivalen a 1 hora y 5 minutos, podemos agregar esa hora al número de horas resultando así 27 horas. Pero de nuevo, 27 horas equivalen a 1 día y 3 horas, y hemos podido agregar ese día al número de días resultando así 4 días. Por lo que el resultado válido debe ser: 4 días, 3 horas, 5 minutos, 30 segundos

Por ejemplo: Si la cantidad es 309.639 segundos entonces el resultado de nuestro programa deberá ser: 3 días, 14 horas, 0 minutos y 39 segundos.

Mejoremos el enunciado anterior, hagamos una especificación adecuada del programa. Denotemos por **N** el número de segundos que queremos convertir en días, horas, minutos y segundos. Suponemos que **N** es un número entero no negativo. Los resultados que queremos produzca el programa son cuatro números enteros no negativos. Para ello debemos declarar cuatro variables tipo entero **d, h, m, s**, que corresponderán respectivamente a los días, los minutos, las horas y los segundos. La relación que debe existir entre **N** y **d, m, h, m, s** es la siguiente:

$$\mathbf{N = 86400*d + 3.600*h + 60*m + s} \\ \wedge \\ \mathbf{0 \leq s < 60 \;\wedge\; 0 \leq m < 60 \;\wedge\; 0 \leq h < 24 \;\wedge\; 0 \leq d}$$

donde 86.400, 3.600, 60 son los números de segundos que posee, respectivamente un día, una hora y un minuto. Por lo tanto la especificación del programa es:

[**const** N: entero
 var d, h, m, s: entero;
 { N ≥ 0 }
 S
 {(N = 86.400*d + 3.600*h + 60*m + s) ∧
 0 ≤ s < 60 ∧ 0 ≤ m < 60 ∧ 0 ≤ h < 24 ∧ 0 ≤d }
].

La manera, conocida por todos, de calcular **d** es determinando el cociente de la división entera de **N** entre 86.400 (esto se debe al hecho que 3.600***h** + 60***m** + **s** es menor estricto que 86400 y a la definición de división entera. Muestre que según este hecho d es único, así como serán únicos h, m, y s). El resto de esa división nos dará el número de segundos restantes. Ahora esos segundos restantes los podremos convertir en horas, minutos y segundos procediendo en forma similar. Note que estamos aplicando diseño

descendente en el razonamiento anterior: el problema lo dividimos en dos sub-problemas más simples de resolver, a saber, calcular d y luego calcular h, m, s.

Nuestro problema se reduce a determinar S1 y S2 tal que se cumpla:

$\{ P: N \geq 0 \}$
S1
$\{ Q: N = 86.400 * d + rd \quad \wedge \quad 0 \leq rd < 86.400 \ \wedge \ 0 \leq d \}$
S2
$\{ R: (N = 86.400*d + 3.600*h + 60*m + s) \wedge 0 \leq s < 60 \wedge 0 \leq m < 60 \wedge 0 \leq h < 24 \wedge$
$0 \leq d \}$

Proponemos que S1 sea:

d := N div 86.400;
rd := N mod 86.400

Mostremos que se cumple {P} S1 {Q}:

Q(rd := N mod 86.400)
≡ regla de sustitución
$Q1: N = 86.400 * d + N \bmod 86.400 \quad \wedge \quad 0 \leq N \bmod 86.400 < 86.400 \ \wedge \ 0 \leq d$

El predicado Q1 lo tomamos como postcondición de la instrucción d := N div 86.400 y vemos que:

Q1 (d := N div 86.400)
≡ regla de sustitución
$N = 86.400 * (N \text{ div } 86.400) + N \bmod 86.400 \quad \wedge \quad 0 \leq N \bmod 86.400 < 86.400$
$\wedge \quad 0 \leq N \text{ div } 86.400$
⇐ por definición de div y mod
$N \geq 0$

Por la regla de secuenciación tenemos que {P} S1 {Q} se cumple.

Ahora el problema restante lo podemos dividir en dos sub-problemas: calcular el número de horas **h,** y luego, calcular **m** y **s**. El número de horas **h** lo podemos calcular de manera similar a como calculamos **d**, pero a partir de **rd,** que representa la cantidad de segundos restantes. Si vemos la especificación original, se requiere que **h** sea menor que 24.

La especificación de este trozo de programa sería:

$\{ Q: N = 86.400 * d + rd \quad \wedge \quad 0 \leq rd < 86.400 \ \wedge \ 0 \leq d \}$
S3
$\{ Q2: (N = 86.400*d + 3.600*h + rh) \wedge 0 \leq rh < 3.600 \wedge 0 \leq h < 24 \wedge 0 \leq d \}$

Así **rd** = 3.600 * **h** + **rh**, con **h** ≥ 0 y 0 ≤ **rh** < 3.600, donde **rh** representa la cantidad de segundos restantes después de haber calculado los días (**d**) y las horas (**h**). Proponemos como S3:

h := rd div 3.600;
rh := rd mod 3.600

Mostremos que S3 cumple la especificación anterior:

Q2 (rh := rd mod 3.600)
≡ regla de sustitución
(N = 86.400*d + 3.600*h + rd mod 3.600) ∧ 0 ≤ rd mod 3.600 < 3.600
∧ 0 ≤ h < 24 ∧ 0 ≤ d
⇐ definición de mod
Q3: (N = 86.400*d + 3.600*h + rd mod 3.600) ∧ 0 ≤ rd ∧ 0 ≤ h < 24 ∧ 0 ≤ d

Tomando Q3 como postcondición de h := rd div 3.600, tenemos:

Q3 (h := rd div 3.600)
≡ regla de sustitución
(N = 86.400*d + 3.600* rd div 3.600 + rd mod 3.600) ∧ 0 ≤ rd
∧ 0 ≤ rd div 3.600 < 24 ∧ 0 ≤ d
⇐ ver demostración más abajo.
N = 86.400 * d + rd ∧ 0 ≤ rd < 86.400 ∧ 0 ≤ d

Si 0 ≤ **rd** < 86.400 entonces se cumple que:

0 ≤ **rd** div 3.600 < 86.400 div 3.600 = 24

Por otro lado, por definición de div y mod:

rd = 3.600* **rd** div 3.600 + **rd** mod 3.600

Aplicando la regla de la secuenciación llegamos a que se cumple:

{ P: N ≥ 0 }
d := N div 86.400;
rd := N mod 86.400;
h := rd div 3.600;
rh := rd mod 3.600
{ Q2: (N = 86.400*d + 3.600*h + rh) ∧ 0 ≤ rh < 3.600 ∧ 0 ≤ h < 24 ∧ 0 ≤ d }

Faltaría determinar el trozo de programa S4 que cumple:

{ Q2 } S4 { R }

Razonando de manera similar obtenemos que S4 es:

```
m := rh div 60;
s := rh mod 60;
```

Entonces el proceso de cálculo sería: se calcula el cociente y el resto de la división entera de N entre 86.400; el cociente se asigna a d y el resto a una nueva variable que denominamos rd, pues necesitamos este valor para continuar con el cálculo de h, m y s. Para calcular h, calculamos primero el cociente y el resto de dividir rd entre 3.600; el cociente será h y el resto lo asignamos a una nueva variable que llamaremos rh. Finalmente m y s serán respectivamente el cociente y resto de la división entera de rh entre 60. El programa completo sería:

```
[ const N: entero
  var d, rd, h, rh, m, s: entero;
  { N ≥ 0 }
   d := N div 86.400;
   rd := N mod 86.400;
   h := rd div 3.600;
   rh := rd mod 3.600;
   m := rh div 60;
   s := rh mod 60;
  { R:( N = 86.400*d + 3.600*h + 60*m + s ) ∧ 0 ≤ s < 60 ∧ 0≤m< 60 ∧ 0 ≤h <24 ∧ 0 ≤d
}
].
```

Ejercicios:

1) Demuestre la correctitud del programa anterior (la solución de este ejercicio mostrará la conveniencia de desarrollar el programa paso a paso paralelamente a la demostración de su correctitud y utilizando análisis desdencente, tal y como fue expuesto antes).
2) Resolver el mismo problema anterior pero calculando primero los minutos luego las horas y luego los días.

Solución al ejercicio (1) anterior:

Partiendo de la postcondición y aplicando sustitución tenemos:

Q(s := rh mod 60)
≡ sustitución de s por rh mod 60
(N = 86.400*d + 3.600*h + 60*m + rh mod 60) ∧ 0 ≤ rh mod 60 < 60 ∧ 0≤m< 60 ∧ 0 ≤ h <24 ∧ 0 ≤d
≡ por definición de mod
Q1: (N = 86.400*d + 3.600*h + 60*m + rh mod 60) ∧ 0≤m< 60 ∧ 0 ≤ rh ∧ 0 ≤ h <24 ∧ 0 ≤d

Q1(m := rh div 60)
≡ sustitución de m por (rh div 60)
Q2: (N = 86.400*d + 3.600*h + 60*(rh div 60) + rh mod 60) ∧ 0≤ rh div 60 < 60 ∧
0 ≤ rh ∧ 0 ≤ h <24 ∧ 0 ≤d

Q2(rh := rd mod 3.600)
≡ sustitución de rh por (rd mod 3.600)
(N = 86.400*d + 3.600*h + 60*((rd mod 3.600) div 60) + ((rd mod 3.600) mod 60)) ∧
0≤ (rd mod 3.600) div 60 < 60 ∧ 0 ≤ (rd mod 3.600) ∧ 0 ≤ h <24 ∧ 0 ≤ d
≡ simplificación usando definición de mod y div
Q3: (N = 86.400*d + 3.600*h + 60*((rd mod 3.600) div 60) + (rd mod 3.600) mod 60)
∧ 0≤ rd ∧ 0 ≤ h <24 ∧ 0 ≤ d

Q3 (h := rd div 3.600)
≡ sustitución de h por (rd mod 3.600)
(N = 86.400*d + 3.600*(rd div 3.600) + 60*((rd mod 3.600) div 60) +
(rd mod 3.600) mod 60) ∧ 0≤ rd ∧ 0 ≤ (rd div 3.600) <24 ∧ 0 ≤ d
≡ 0 ≤ rd es redundante
Q4: (N = 86.400*d + 3.600*(rd div 3.600) + 60*((rd mod 3.600) div 60) +
(rd mod 3.600) mod 60) ∧ 0 ≤ (rd div 3.600) < 24 ∧ 0 ≤ d

Q4 (rd := N mod 86.400)
≡ sustitución de rd por (N mod 86.400)
(N = 86.400*d + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60) ∧ 0 ≤ ((N mod 86.400) div 3.600) < 24 ∧ 0 ≤ d
≡ como 86.400 = 24*3.600 se tiene que 0 ≤ ((N mod 86.400) div 3.600) < 24 ≡ n≥ 0
Q5: (N = 86.400*d + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60) ∧ 0 ≤ N ∧ 0 ≤ d

Q5 (d := N div 86.400)
≡ sustitución de d por (N div 86.400)
(N = 86.400*(N div 86.400) + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60) ∧ 0 ≤ N ∧ 0 ≤ (N div 86.400)
≡ simplificación
(N = 86.400*(N div 86.400) + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60) ∧ 0 ≤ N
≡ definición de mod y div (ver justificación más abajo)
0 ≤ n

Probar que la siguiente igualdad siempre se cumple para N entero no negativo:

(1) N = 86.400*(N div 86.400) + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60

En efecto a N lo podemos escribir en la forma:
N = 86.400 * q + r donde q, r representan respectivamente el cociente y resto de la división entera de N entre 86.400

Así q es igual a (N div 86.400) y r es (N mod 86.400) por definición de div y mod

Por lo que probar:

N = 86.400*(N div 86.400) + 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60

es equivalente a probar:

(N mod 86.400) = 3.600*((N mod 86.400) div 3.600) + 60*(((N mod 86.400) mod 3.600) div 60) + ((N mod 86.400) mod 3.600) mod 60

esta igualdad sería consecuencia de probar:

r = 3.600*(r div 3.600) + 60*((r mod 3.600) div 60) + (r mod 3.600) mod 60, $r \geq 0$

Aplicando un razonamiento similar a lo ya expuesto, obtendremos que la igualdad (1) es verdad cualquiera sea N entero no negativo.

4.3. Acciones Parametrizadas: Procedimientos y Funciones

4.3.1. Procedimientos

Cuando diseñamos un algoritmo para resolver un problema de cierta complejidad, la técnica de diseño descendente nos dice que es conveniente descomponer el problema en varios subproblemas correctamente especificados. Un algoritmo que resuelva el problema original resultará de ensamblar correctamente (por ejemplo, secuencialmente) las acciones "abstractas" que resuelven cada uno de los subproblemas. Si cada subproblema es especificado correctamente y sus soluciones corresponden a algoritmos correctos, entonces el problema original quedará resuelto correctamente. En otras palabras, para entender lo que hace un algoritmo y probar su correctitud, basta con conocer y especificar precisamente *lo que hacen* cada una de las acciones que lo conforman sin necesidad de saber *cómo lo hacen*.

En el proceso de convertir el algoritmo en un programa, las acciones "abstractas" (el término *abstracto* es utilizado aquí para indicar que las acciones vienen expresadas en términos de la información descrita por el enunciado original del problema y no en términos de las instrucciones del lenguaje de programación que se vaya a utilizar) son refinadas a niveles más bajos de abstracción, convirtiéndose en secuencias de otras

acciones menos abstractas. Como en cada nivel de abstracción, a cada acción "abstracta" la denotamos por un nombre que designa lo que ella hace, podemos decir que en el proceso de desarrollo de programas, vamos dando nombres (el nombre de la acción abstracta) a la secuencia de acciones resultante del refinamiento de cada acción abstracta. Esta secuencia de acciones con nombre corresponderá a un nuevo constructor de nuestro pseudolenguaje que llamaremos *procedimiento o método*.

Los procedimientos son una herramienta muy útil cuando queremos controlar la complejidad de las soluciones algorítmicas de un problema, porque el programa final no será una secuencia grande y monolítica de instrucciones del lenguaje de programación, sino que éste estará estructurado de acuerdo a los niveles de abstracción en que fuimos refinando la solución hasta llegar al programa. El programa original, estructurado de esta forma, contendrá entre sus instrucciones, algunas instrucciones (*llamadas a procedimientos*) que servirán para indicar que se ejecutarán algunos procedimientos y estos procedimientos a su vez contendrán algunas instrucciones de este tipo que servirán para indicar que se ejecutarán otros procedimientos, y así sucesivamente. En este sentido una utilidad importante de los procedimientos es permitir expresar niveles de abstracción y un medio que permite mantener el control de la complejidad en la programación.

Por otro lado, puede pasar que en distintas partes de un mismo programa se quiera ejecutar la misma secuencia de acciones pero para distintos estados iniciales. Si se tiene el constructor *procedimiento (o método)*, entonces en cada punto del programa donde se tenga que ejecutar el mismo conjunto de acciones, podremos reemplazar ese conjunto de acciones por el nombre del procedimiento y el estado inicial adecuado, lo cual reduce el número de instrucciones del programa. El conjunto de acciones entonces *se declara* (se define) aparte asignándole un nombre (el nombre del procedimiento o método); habrá que parametrizar la secuencia de acciones, de manera que pueda ser ejecutada desde distintos puntos del programa con distintos estados iniciales.

Veamos un ejemplo para aclarar los comentarios hechos hasta ahora:

Problema: Determinar el equivalente en días, horas, minutos y segundos de tres números enteros X, Y, Z que representan segundos.

Primera Solución:

La especificación del problema sería:

[const X, Y, Z: entero
  var dx, hx, mx, sx: entero;
  var dy, hy, my, sy: entero;
  var dz, hz, mz, sz: entero;
  { $X \geq 0, Y \geq 0, Z \geq 0$ }

  S

$\{$ (X = 86.400*dx + 3.600*hx + 60*mx + sx) $\wedge$ 0 ≤ sx < 60 $\wedge$ 0≤mx< 60 $\wedge$ 0 ≤hx <24
$\wedge$ 0 ≤dx $\wedge$
(Y = 86.400*dy + 3.600*hy + 60*my + sy) $\wedge$ 0 ≤ sy < 60 $\wedge$ 0≤my< 60 $\wedge$ 0 ≤hy <24
$\wedge$ 0 ≤dy $\wedge$
(Z = 86.400*dz + 3.600*hz + 60*mz + sz) $\wedge$ 0 ≤ sz < 60 $\wedge$ 0≤mz< 60 $\wedge$ 0 ≤hz <24
$\wedge$ 0 ≤dz $\}$
].

Aplicando diseño descendente, el problema original se puede dividir en tres subproblemas: calcular los días, horas, minutos y segundos de X, luego calcular los días, horas, minutos y segundos de Y, y finalmente calcular los días, horas, minutos y segundos de Z.

Para resolver cada uno de los subproblemas anteriores podemos utilizar el programa de la sección 4.2, que sabemos es correcto, para calcular los días, horas, minutos y segundos correspondientes a una cantidad dada de segundos. Lo que habría que hacer es tomar el programa, reemplazar el nombre a las variables n, d, h, m, y s, respectivamente por X, dx, hx, mx y sx, para que calcule los días, horas, minutos y segundos correspondientes a X según la especificación anterior. Esto resulta en el trozo de programa siguiente, que llamaremos S1:

```
dx := X div 86.400;
rd := X mod 86.400;
hx := rd div 3.600;
rh := rd mod 3.600;
mx := rh div 60;
sx := rh mod 60
```

Luego tomamos el mismo programa de la sección 4.2. y reemplazamos el nombre a las variables n, d, h, m, y s, respectivamente por Y, dy, hy, my y sy, para que calcule los días, horas, minutos y segundos correspondientes a Y. Esto resulta en el trozo de programa siguiente, que llamaremos S2:

```
dy := Y div 86.400;
rd := Y mod 86.400;
hy := rd div 3.600;
rh := rd mod 3.600;
my := rh div 60;
sy := rh mod 60
```

Finalmente, tomamos el mismo programa de la sección 4.2. y reemplazamos el nombre a las variables n, d, h, m, y s, respectivamente por Z, dz, hz, mz y sz, para que calcule los días, horas, minutos y segundos correspondientes a Z. Esto resulta en el trozo de programa siguiente, que llamaremos S3:

```
dz := Z div 86.400;
```

```
   rd := Z mod 86.400;
   hz := rd div 3.600;
   rh := rd mod 3.600;
   mz := rh div 60;
   sz := rh mod 60
```

Ensamblando la solución del problema original a partir de la solución de cada subproblema nos queda el programa:

```
[ const X, Y, Z: entero;
  var rd, rh: entero;
  var dx, hx, mx, sx: entero;
  var dy, hy, my, sy: entero;
  var dz, hz, mz, sz: entero;

  { X ≥ 0, Y ≥ 0, Z ≥ 0 }

   dx := X div 86.400;
   rd := X mod 86.400;
   hx := rd div 3.600;
   rh := rd mod 3.600;
   mx := rh div 60;
   sx := rh mod 60;
   dy := Y div 86.400;
   rd := Y mod 86.400;
   hy := rd div 3.600;
   rh := rd mod 3.600;
   my := rh div 60;
   sy := rh mod 60;
   dz := Z div 86.400;
   rd := Z mod 86.400;
   hz := rd div 3.600;
   rh := rd mod 3.600;
   mz := rh div 60;
   sz := rh mod 60

  { (X = 86.400*dx + 3.600*hx + 60*mx + sx) ∧ 0 ≤ sx < 60 ∧ 0≤mx< 60 ∧ 0 ≤hx <24
      ∧ 0 ≤dx  ∧
    (Y = 86.400*dy + 3.600*hy + 60*my + sy) ∧ 0 ≤ sy < 60 ∧ 0≤my< 60 ∧ 0 ≤hy <24
      ∧ 0 ≤dy  ∧
     (Z = 86.400*dz + 3.600*hz + 60*mz + sz) ∧ 0 ≤ sz < 60 ∧ 0≤mz< 60 ∧ 0 ≤hz <24
       ∧ 0 ≤dz    }
].
```

Aparentemente el programa anterior resuelve nuestro problema original, sin embargo note que no hemos probado que se cumple lo siguiente:

$\{ X \geq 0, Y \geq 0, Z \geq 0 \}$
S1;
S2;
S3
$\{ (X = 86.400*dx + 3.600*hx + 60*mx + sx) \wedge 0 \leq sx < 60 \wedge 0 \leq mx < 60 \wedge 0 \leq hx < 24 \wedge 0 \leq dx \wedge$
$(Y = 86.400*dy + 3.600*hy + 60*my + sy) \wedge 0 \leq sy < 60 \wedge 0 \leq my < 60 \wedge 0 \leq hy < 24 \wedge 0 \leq dy \wedge$
$(Z = 86.400*dz + 3.600*hz + 60*mz + sz) \wedge 0 \leq sz < 60 \wedge 0 \leq mz < 60 \wedge 0 \leq hz < 24 \wedge 0 \leq dz \}$

Es decir, hay que probar que la secuenciación de S1, S2 y S3 (ese ensamblaje!) cumple con la especificación del problema original. Intuitivamente vemos que esto es verdad ya que las variables que intervienen en S1 no son modificadas por S2 y las variables que intervienen en S1 y en S2 no son modificadas por S3, así que lo calculado por S1, S2 y S3 por separado, no se altera si ejecutamos la secuencia S1; S2; S3. Para probar formalmente este hecho utilizamos la regla siguiente, que llamaremos *regla de fortalecimiento*:

Si $\{P\}$ S $\{Q\}$ se cumple y R es un predicado cuyas variables libres no aparecen en S entonces $\{P \wedge R\}$ S $\{Q \wedge R\}$ se cumple.

En nuestro caso tenemos que originalmente se cumple lo siguiente:

- $\{ X \geq 0 \}$ S1 $\{ R1: (X = 86.400*dx + 3.600*hx + 60*mx + sx) \wedge 0 \leq sx < 60 \wedge 0 \leq mx < 60 \wedge 0 \leq hx < 24 \wedge 0 \leq dx \}$

- $\{ Y \geq 0 \}$ S2 $\{ R2: (Y = 86.400*dy + 3.600*hy + 60*my + sy) \wedge 0 \leq sy < 60 \wedge 0 \leq my < 60 \wedge 0 \leq hy < 24 \wedge 0 \leq dy \}$

- $\{ Z \geq 0 \}$ S3 $\{ R3: (Z = 86.400*dz + 3.600*hz + 60*mz + sz) \wedge 0 \leq sz < 60 \wedge 0 \leq mz < 60 \wedge 0 \leq hz < 24 \wedge 0 \leq dz \}$

Tomando a R, en la regla de fortalecimiento, como $Y \geq 0 \wedge Z \geq 0$, se cumple entonces:

$\{ Q1: X \geq 0 \wedge \quad Y \geq 0 \wedge Z \geq 0\}$

S1

$\{ Q2: \; R1 \wedge \quad Y \geq 0 \wedge Z \geq 0\}$

Tomando a R, en la regla de fortalecimiento, como:

$Z \geq 0 \wedge R1$

se cumple:

{ Q2: $Y \geq 0 \wedge Z \geq 0 \wedge R1$ }

S2

{ Q3: $R2 \wedge Z \geq 0 \wedge R1$ }

Tomando a R, en la regla de fortalecimiento, como:

$$R1 \wedge R2$$

se cumple:

{ Q3: $Z \geq 0 \wedge R1 \wedge R2$ }

S3

{ Q4: $R1 \wedge R2 \wedge R3$ }

Finalmente por la regla de la secuenciación se prueba formalmente que el programa es correcto, pues al cumplirse:

{Q1} S1; {Q2} S2; {Q3} S3 {Q4}

se cumple:

{Q1} S1; S2; S3 {Q4}

Segunda Solución:

El algoritmo anterior lo podemos escribir en forma muy compacta, de manera que refleje el "pequeño" diseño descendente que se hizo. Si tuviéramos una instrucción (o acción) como la siguiente:

Convertir (n, d, h, m, s)

cuyo significado operacional es "convertir la cantidad de segundos **n** en días, horas, minutos y segundos, según la especificación dada en la sección 4.2., y guardar estos valores respectivamente en **d**, **h**, **m** y **s**". Entonces, el programa para convertir tres cantidades X, Y, Z de segundos en días, horas, minutos y segundos se reduciría a:

```
[ const X, Y, Z: entero;
  var dx, hx, mx, sx: entero;
```

```
  var dy, hy, my, sy: entero;
  var dz, hz, mz, sz: entero;

  { X ≥ 0, Y ≥ 0, Z ≥ 0 }

  Convertir(X, dx, hx, mx, sx);
  Convertir(Y, dy, hy, my, sy);
  Convertir(Z, dz, hz, mz, sz)

  { (X = 86.400*dx + 3.600*hx + 60*mx + sx) ∧ 0 ≤ sx < 60 ∧ 0≤mx< 60 ∧ 0 ≤hx <24
     ∧ 0 ≤dx  ∧
    (Y = 86.400*dy + 3.600*hy + 60*my + sy) ∧ 0 ≤ sy < 60 ∧ 0≤my< 60 ∧ 0 ≤hy <24
     ∧ 0 ≤dy  ∧
    (Z = 86.400*dz + 3.600*hz + 60*mz + sz) ∧ 0 ≤ sz < 60 ∧ 0≤mz< 60 ∧ 0 ≤hz <24
     ∧ 0 ≤dz    }
].
```

Note que dimos un nombre, "**Convertir**", al conjunto de acciones que calcula los días, horas, minutos y segundos de una cantidad dada en segundos; además, parametrizamos ese conjunto de acciones para que pueda ser ejecutado con variables de diferentes nombres; en el ejemplo anterior, primero con X, dx, hx, mx, sx, luego con Y, dy, hy, my, sy, y finalmente con Z, dz, hz, mz, sz. El conjunto de acciones, junto con el nombre y los parámetros, es lo que llamaremos *procedimiento o método*.

En el pseudolenguaje, el procedimiento (o método) "Convertir" lo definimos (o *declaramos*) de la siguiente manera:

proc Convertir (**entrada** n: entero; **salida** d, h, m, s : entero)

```
[   var rd, rh: entero;

  { n = N, N ≥ 0 }

   d := n div 86.400;
   rd := n mod 86.400;
   h := rd div 3.600;
   rh := rd mod 3.600;
   m := rh div 60;
   s := rh mod 60

  { (N = 86.400*d + 3.600*h + 60*m + s) ∧ 0 ≤ s < 60 ∧ 0≤m< 60 ∧ 0 ≤h <24
     ∧ 0 ≤d     }
].
```

donde **proc**, **entrada** y **salida** son palabras reservadas del pseudolenguaje para definir un procedimiento. A las variables entre paréntesis en la cabecera del procedimiento se les

llama *parámetros formales*.

Un procedimiento se ejecuta mediante una instrucción *de llamada al procedimiento*, por ejemplo, Convertir(X, dx, hx, mx, sx). Esta instrucción se denota colocando el nombre del procedimiento y entre paréntesis las variables sobre las cuales se ejecutará el procedimiento. Cada una de estas variables, llamadas *parámetros reales o argumentos,* deberá ser del mismo tipo que el parámetro formal correspondiente. La manera formal de demostrar la correctitud del programa anterior (secuencia de tres llamadas a procedimientos) la veremos más adelante cuando definamos formalmente lo que significa una llamada a un procedimiento.

Como vemos en el ejemplo anterior, usar un procedimiento es como usar cualquier operación elemental del pseudolenguaje (por ejemplo, +), sabemos lo que hace la suma pero no cómo se lleva a cabo. Al escribir un procedimiento estamos prácticamente extendiendo el lenguaje para que incluya otras operaciones. Por ejemplo, cuando usamos + en una expresión, nunca nos preguntamos cómo esta es ejecutada; asumimos que + funciona correctamente. De manera similar, cuando escribimos una llamada a un procedimiento, lo que nos interesa es "qué" hace el procedimiento y no "cómo" lo hace, y confiamos en que lo que se supone que hace, lo hace correctamente. Podemos entonces ver un procedimiento (y la demostración de su correctitud) como un "lema". Un programa puede ser considerado como una demostración constructiva de que su especificación es consistente y calculable; y un procedimiento es un lema usado en la demostración constructiva.

En general un procedimiento en el pseudolenguaje se declarará de la siguiente forma:

{ Pre: P}
{ Post: Q }
proc <identificador> (<especificación de parámetros>; ... ;
<especificación de parámetros>)
<cuerpo del procedimiento>

donde cada <especificación de parámetros> tiene una de las tres formas siguientes:

entrada <lista de identificadores> : <tipo>
entrada-salida <lista de identificadores> : <tipo>
salida <lista de identificadores> : <tipo>

<lista de identificadores> corresponde a una lista de uno o más identificadores de variables separados por comas, como por ejemplo: x, y, z

Cada identificador es un parámetro formal. Cada lista de identificadores tiene asociado un tipo (entero, booleano, etc.), el cual define el tipo que deberá tener el correspondiente parámetro real (o argumento) en una llamada al procedimiento.

<cuerpo del procedimiento> es un programa, como ya lo hemos definido en el

pseudolenguaje sin incluir la pre y post condiciones, P y Q, que serán colocadas antes del encabezado del procedimiento. La ejecución de una llamada a un procedimiento será la ejecución del programa <cuerpo del procedimiento>. Durante la ejecución de <cuerpo del procedimiento>, los parámetros son considerados variables locales (es decir, sólo son válidos en el cuerpo del procedimiento y no fuera de él. Decimos también que su *alcance* es el cuerpo del procedimiento <cuerpo del procedimiento>).

Los valores iniciales de los parámetros y el uso de sus valores finales están determinados por los atributos **entrada**, **entrada-salida**, **salida,** dados a los parámetros en el encabezado del procedimiento. Hablaremos respectivamente de *parámetros de entrada*, *parámetros de entrada-salida* y *parámetros de salida*. Los datos de entrada al procedimiento vienen dados por los parámetros de entrada y los parámetros de entrada-salida. Los parámetros de entrada no pueden ser modificados en el cuerpo del procedimiento, es decir, se comportan como variables declaradas con atributo "const". Antes de la ejecución del procedimiento, estos parámetros se inicializan con los valores proporcionados por una llamada al procedimiento. Los resultados de la ejecución del procedimiento serán almacenados en los parámetros de entrada-salida y salida, para luego ser copiados sus valores en los argumentos correspondientes en una llamada al procedimiento.

Suponemos que se cumple { P } <cuerpo del procedimiento> { Q }, y que esta información será utilizada cuando escribamos llamadas al procedimiento. Cuando declaremos un procedimiento, escribiremos la precondición y la postcondición antes del cuerpo del procedimiento para que se haga más fácil encontrar la información necesitada al momento de escribir y entender una llamada a un procedimiento. Para escribir una llamada a un procedimiento sólo necesitamos entender las líneas:

{ Pre: P}
{ Post: Q }
proc <identificador> (<especificación de parámetros>; ... ;
<especificación de parámetros>)

Las pre y post condiciones indican el "qué hace" el procedimiento y el cuerpo es el "cómo lo hace". Por lo tanto una especificación de un procedimiento vendrá dada por la precondición, la postcondición y el encabezado donde se menciona el nombre del procedimiento y se declaran los parámetros formales.

Con el propósito de garantizar la correctitud de los programas que haremos, impondremos las siguientes restricciones en el uso de identificadores (nombres de variables ó nombres de procedimientos) en una declaración de un procedimiento: los únicos identificadores que pueden ser utilizados en el cuerpo del procedimiento son los parámetros, las variables declaradas en el cuerpo en sí y otros procedimientos (en términos de programación esto significa que no se permitirán "variables globales", pero por el momento no nos interesará saber lo que significa "variable globales"). Los parámetros deben ser identificadores distintos entre sí. La precondición P sólo debe

contener como variables libres los parámetros con atributo **entrada**, **entrada-salida** y variables de especificación. La postcondición Q puede contener como variables libres los parámetros de **entrada**, **salida**, **entrada-salida**, y variables de especificación.

La declaración del procedimiento Convertir sería:

proc Convertir (**entrada** n: entero; **salida** d, h, m, s : entero)
{ **Pre**: $n = N \land N \geq 0$ }
{ **Post**: $(N = 86.400*d + 3.600*h + 60*m + s) \land 0 \leq s < 60 \land 0 \leq m < 60 \land 0 \leq h < 24 \land 0 \leq d$ }

```
[   var rd, rh: entero;
   d := n div 86.400;
   rd := n mod 86.400;
   h := rd div 3.600;
   rh := rd mod 3.600;
   m := rh div 60;
   s := rh mod 60
].
```

Note que como el parámetro de entrada n no puede ser modificado en el cuerpo del procedimiento podríamos haber prescindido de la variable de especificación N. Sin embargo utilizamos una variable de especificación N para enfatizar el hecho que en la postcondición nos referimos al valor original de n.

Definición formal de una llamada a un procedimiento:

En lo que sigue supondremos que un procedimiento tiene la forma siguiente:

proc p(**entrada** $\underline{x}$; **entrada-salida** $\underline{y}$; **salida** $\underline{z}$)
{ P }
{ Q }
S

donde $\underline{x}$ representa la lista de los parámetros x_i de entrada del procedimiento p, $\underline{y}$ representa la lista de los parámetros y_i de entrada-salida y $\underline{z}$ la lista de los parámetros z_i de salida. Los identificadores de los parámetros deben ser distintos entre si. No tomamos en cuenta los tipos pues no intervienen en lo que expondremos a continuación.

Estamos interesados ahora en definir formalmente la instrucción "llamada a un procedimiento" en nuestro pseudolenguaje, la cual tiene la forma:

$$p(\underline{a}, \underline{b}, \underline{c})$$

El nombre del procedimiento es p. $\underline{a}$, $\underline{b}$, $\underline{c}$ son las respectivas listas de los parámetros reales o argumentos a_i, b_i, c_i separados por comas. Los a_i son expresiones (por ejemplo, t

ó t*s), mientras que los b_i y los c_i son variables. Los a_i , b_i , c_i son los argumentos de entrada, entrada-salida, y salida, que corresponden respectivamente a los parámetros formales x_i , y_i , z_i de las listas $\underline{x}$, $\underline{y}$, $\underline{z}$ .Cada argumento debe ser del mismo tipo que su parámetro formal correspondiente. Si un argumento es una constante, sólo podrá corresponder a un parámetro formal de entrada.

Los identificadores, sean nombres de variables o procedimientos, que están accesibles (puedan ser usados) al momento de la llamada al procedimiento, deben ser diferentes de los parámetros formales $\underline{x}$, $\underline{y}$, $\underline{z}$ del procedimiento. Esta restricción evita tener que usar notación extra para manejar el conflicto que se presenta al tener un mismo identificador utilizado con dos propósitos diferentes, pero no es esencial.

Una llamada al procedimiento Convertir del ejemplo dado antes sería: Convertir(X, dh, hx, mx, sx). Su ejecución convierte la cantidad de segundos X en días, horas, minutos y segundos, según la especificación dada en la sección 4.2., y almacena estos valores respectivamente en dx, hx, mx y sx.

En general, la interpretación operacional de la llamada p($\underline{a}$, $\underline{b}$, $\underline{c}$) es como sigue:

> Todos los parámetros son considerados variables locales del procedimiento (su alcance es el cuerpo del procedimiento). Primero, se determina los valores de los argumentos de entrada y entrada-salida $\underline{a}$ y $\underline{b}$ y se almacenan en los parámetros correspondientes $\underline{x}$, $\underline{y}$. Segundo, se determina la dirección en memoria de los argumentos de entrada-salida y salida $\underline{b}$, $\underline{c}$. Note que todos los parámetros con atributo **entrada** son inicializados, mientras que los otros no. Tercero, se ejecuta el cuerpo del procedimiento. Cuarto, se almacena los valores de los parámetros de entrada-salida y salida $\underline{y}$, $\underline{z}$ en los correspondientes argumentos de entrada-salida y salida $\underline{b}$, $\underline{c}$ (utilizando las direcciones de memoria previamente determinadas) *en orden de izquierda a derecha (esto es necesario precisarlo debido a que un parámetro real de salida puede aparecer más de una vez en la lista de parámetros de una llamada).*

Para definir formalmente una llamada a procedimiento, definamos primero lo que significa la instrucción de asignación múltiple, que tiene la forma siguiente:

$$x_1, x_2, \ldots , x_n := e_1, e_2, \ldots , e_n$$

La interpretación operacional de la instrucción anterior es la siguiente: Primero, evalúe las expresiones e_i, para todo i entre 1 y n. Sea v_i el valor resultante de evaluar e_i. Luego asigne primero v_1 a x_1, luego v_2 a x_2 , ... , y finalmente v_n a x_n. Note que para que un predicado Q se cumpla después de la asignación múltiple anterior entonces antes de esta asignación se debe cumplir el predicado que resulta de sustituir cada ocurrencia de x_i en Q por e_i, para todo i entre 1 y n. Este predicado lo podemos denotar por $Q(\underline{x}:=\underline{e})$, donde $\underline{x}$, $\underline{e}$ son respectivamente las listas de las variables x_i y las expresiones e_i.

La definición formal de una llamada a procedimiento es como sigue. De la anterior

interpretación operacional, vemos que la ejecución de la llamada p($\underline{a}$, $\underline{b}$, $\underline{c}$) es equivalente a la ejecución de la secuencia:

$$\underline{x}, \underline{y} := \underline{a}, \underline{b} \; ; \; S \; ; \; \underline{b}, \underline{c} \; := \; \underline{y}, \underline{z}$$

(las direcciones de memoria de $\underline{b}$, $\underline{c}$ pueden ser evaluadas antes o después de la ejecución de S, ya que la ejecución de S no modifica sus valores).

Regla de llamada a procedimiento:

Suponiendo que {P} S {Q} se cumple, probar formalmente que se cumple:

$$\{ P1 \} \; p(\underline{a}, \underline{b}, \underline{c}) \; \{ Q1 \}$$

Es equivalente a probar que se cumple:

$$\{ P1 \} \; \underline{x}, \underline{y} := \underline{a}, \underline{b} \; ; \{ P \} \; S \; \{ Q \} \; ; \; \underline{b}, \underline{c} \; := \; \underline{y}, \underline{z} \; \{ Q1 \}$$

Y esto a su vez es equivalente a probar:

1) $P1 \Rightarrow P(\underline{x}, \underline{y} := \underline{a}, \underline{b})$ es una tautología.
2) $P1' \wedge Q(\underline{x} := \underline{a}) \Rightarrow Q1(\underline{b}, \underline{c} := \underline{y}, \underline{z})$ es una tautología.

Donde P1 es de la forma $P1 \equiv P1' \wedge R$ y P1' es la parte de P1 que no contiene parámetros de salida ni de entrada-salida. Se dbe colocar $Q(\underline{x} := \underline{a})$ y no al revés pues $\underline{a}$ puede ser una constante o una expresión.

Por principio, Q no debería incluir parámetros de entrada ya que éstos son considerados variables cuyo valor sólo es utilizado en el cuerpo del procedimiento (decimos que son *variables locales* del procedimiento) y por lo tanto no tienen ningún significado una vez concluya la ejecución del cuerpo del procedimiento. Sin embargo, en caso de que $\underline{x}$ aparezca en la postcondición del procedimiento p, hay que hacer el reemplazo de estos parámetros formales de entrada por los reales $\underline{a}$ en la prueba formal de correctitud como muestra (2). Por el contrario, los parámetros de entrada-salida y salida, aunque son locales al procedimiento, contendrán valores que serán utilizados una vez concluya la ejecución del procedimiento y deben aparecer en la postcondición. Es preferible utilizar, en la postcondición de un procedimiento, variables de especificación para denotar el valor inicial de un parámetro de entrada en lugar del parámetro mismo.

Note que (2) es equivalente a probar que $P1' \wedge Q(\underline{x, y}, \underline{z} \; := \; \underline{a, b}, \underline{c}) \Rightarrow Q1$ es una tautología.

Ejemplo 1:

Considere el procedimiento "intercambio" dado por la especificación siguiente:

proc intercambio (entrada-salida y1, y2:entero)
{ Pre: $y1 = X \wedge y2 = Y$ }
{ Post: $y1 = Y \wedge y2 = X$ }

Queremos probar que se cumple:

(1) $\{ a = X \wedge b = Y \}$ intercambio (a, b) $\{ a = Y \wedge b = X \}$

donde a y b son variables enteras y Y, X denotan respectivamente sus valores finales. Aplicando la *regla de llamada a procedimiento* debemos mostrar que los siguientes predicados son tautologías:

- $(a = X \wedge b = Y) \Rightarrow (y1 = X \wedge y2 = Y) (y1, y2 := a,b)$
- $(y1 = Y \wedge y2 = X) \Rightarrow (a = Y \wedge b = X) (a, b := y1, y2)$

lo cual es evidente.

Ejemplo2:

Queremos probar lo siguiente:

$\{ x = X \wedge X \geq 0 \}$

Convertir(x, dx, hx, mx, sx)

$\{ (X = 86.400*dx + 3.600*hx + 60*mx + sx) \wedge 0 \leq sx < 60 \wedge 0 \leq mx < 60 \wedge 0 \leq hx < 24$
$\wedge\ 0 \leq dx \}$

En la especificación del procedimiento **Convertir** podemos reemplazar variables de especificación por otros identificadores nuevos (que no aparezcan en la especificación) y así obtener una especificación equivalente. Reemplazamos N por X:

proc Convertir (**entrada** n: entero; **salida** d, h, m, s : entero)
{ Pre: $n = X \wedge X \geq 0$ }
{ Post: $(X = 86.400*d + 3.600*h + 60*m + s) \wedge 0 \leq s < 60 \wedge 0 \leq m < 60 \wedge 0 \leq h < 24$
$\wedge\ 0 \leq d$ }

Aplicando la *regla de llamada a procedimiento*, tenemos que probar:

1) $(x = X \wedge X \geq 0) \Rightarrow (n = X \wedge X \geq 0) (n:=x)$
2) $((X = 86.400*d + 3.600*h + 60*m + s) \wedge 0 \leq s < 60 \wedge 0 \leq m < 60 \wedge 0 \leq h < 24$
$\wedge\ 0 \leq d) \wedge (x = X \wedge X \geq 0)$
$\Rightarrow$
$((X = 86.400*dx + 3.600*hx + 60*mx + sx) \wedge 0 \leq sx < 60 \wedge 0 \leq mx < 60 \wedge 0 \leq hx$

$<24 \wedge 0 \leq dx)$ (x, dx, hx, mx, sx := n, d, h, m, s)

lo cual resulta evidente.

Este ejemplo ilustra cómo pueden manipularse los valores iniciales y finales de los parámetros en la especificación del procedimiento, con el propósito de hacer la demostración de correctitud. Los identificadores que denotan valores iniciales y finales de los parámetros, como el caso de la variable de especificación N en el ejemplo anterior, pueden ser reemplazados por identificadores nuevos (que no aparezcan en la especificación del procedimiento) para adecuarlos a la demostración formal de la correctitud de la llamada al procedimiento. En el ejemplo reemplazamos N por X en la especificación original.

Ejercicio:

Considere el procedimiento "intercambio" dado por la especificación siguiente:

proc intercambio (entrada-salida y1, y2:entero)
{ Pre: $y1 = X \wedge y2 = Y$ }
{ Post: $y1 = Y \wedge y2 = X$ }

Probar que se cumple:

(1) { $a = A \wedge b = B$ } intercambio (a, b) { $a = B \wedge b = A$ }

Ejemplos sobre uso de procedimientos:

Problema: Hacer un programa que calcule (f(x1)-f(x2))/(x1-x2) para x1 y x2 dados y $f(x) = x^2 + 3x - 2$.

Una manera de razonar para desarrollar nuestro programa, es aplicar el diseño descendente y suponer que tenemos un procedimiento que calcula el valor de f(x) para un x dado. Llamémoslo CalculoDef, cuya llamada sería de la forma CalculoDef(x, z), donde x es un argumento de entrada y representa el valor sobre el cual será evaluada f(x), z representa un argumento de salida cuyo valor será f(x) una vez se ejecute CalculoDef. Utilizando este procedimiento tenemos el programa siguiente para calcular (f(x1)-f(x2)) / (x1-x2) :

```
[ var x1, x2, y1, y2, z : real;
  { x1 = X1 ∧ x2 = X2 ∧ (X1 – X2)≠0 }
  CalculoDef (x1,y1);
  CalculoDef (x2,y2);
  z := (y1 – y2)/(x1 – x2)
  { z = (f(X1)-f(X2)) / (X1-X2) ∧ (∀x: x real: f(x)= x^2 + 3x – 2) }
]
```

El procedimiento CalculoDef sería:

```
proc CalculoDef(entrada x: real; salida z: real)
{Pre: x=X }
{Post: z = X^2 + 3X – 2  }
 [
  z := x*x + 3x – 2
]
```

Hemos podido escribir lo siguiente:

```
proc CalculoDef(entrada x: real; salida z: real)
{Pre: verdad }
{Post: z = x^2 + 3x – 2  }
[
  z := x*x + 3x – 2
]
```

Sin embargo, es preferible utilizar variables de especificación para denotar el valor inicial de x.

Hemos podido hacer el programa sin necesidad de crear el procedimiento CalculoDef, y quedaría de la siguiente forma:

```
[ var x1, x2 : real;
  { x1 = X1 ∧ x2 = X2 ∧ (X1 – X2)≠0 }
   z := ((x1*x1 + 3*x1 –2) – (x2*x2 + 3*x2 –2))/(x1 – x2)
  { z = (f(X1)-f(X2)) / (X1-X2), donde f es la función f(x)= x^2  + 3x - 2 }
]
```

Sin embargo, el programa no refleja la estructura que resulta de la aplicación del diseño descendente, ya que no se hace mención explícita de la función f. La versión que utiliza el procedimiento CalculoDef refleja un mejor estilo de programación.

<u>Ejercicio</u>: Hacer las anotaciones entre cada instrucción de manera que podamos probar que el programa anterior (versión estructurada) es correcto y luego haga la prueba de correctitud formalmente.

<u>Solución parcial</u>:

Algunas aserciones son (faltaría completar y demostrar la correctitud utilizando la regla de fortalecimiento):

```
[ var x1, x2, y1, y2, z : real;
  { x1 = X1 ∧ x2 = X2 ∧ (X1 – X2)≠0 }
  CalculoDef(x1,y1);
```

```
  { y1 = f(X1) }
  CalculoDef(x2,y2);
  { y2 = f(X2) }
  z := (y1 – y2)/(x1 – x2)
  { z = (f(X1)-f(X2)) / (X1-X2), donde f es la función f(x)= x^2 + 3x - 2 }
]
```

Ejercicios:

Desarrolle un procedimiento correcto que dadas la fecha actual y la fecha de nacimiento de una persona, determine la edad en años, meses y días de dicha persona.

Uso de procedimientos: La máquina de trazados

Suponga que contamos con algunos procedimientos (o métodos) que permiten dibujar segmentos de línea recta y círculos en la pantalla del computador, respecto a un eje de coordenadas dado. El origen del eje de coordenadas coincide con el centro de la pantalla, como en la figura 6. El mayor valor absoluto que puede tener la coordenada **x** viene dado por la constante XMAX y el mayor valor absoluto que puede tener la coordenada **y** viene dado por la constante YMAX. Llamaremos a este conjunto de métodos “Máquina de Trazados”.

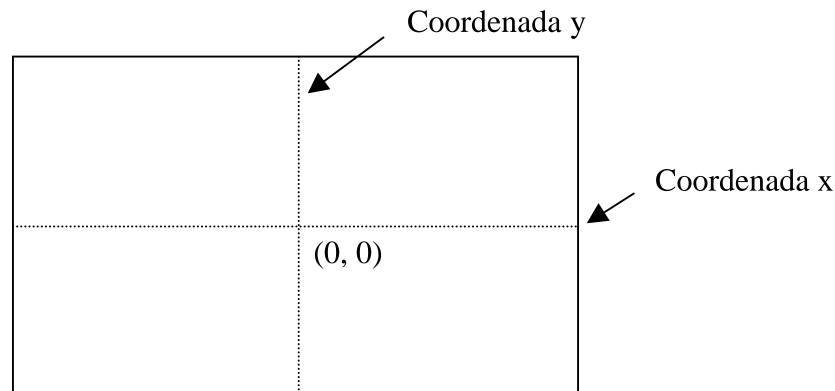

Figura 6

Los métodos son los siguientes:

{ Pre: $|x1| \leq XMAX \wedge |y1| \leq YMAX \wedge |x2| \leq XMAX \wedge |y2| \leq YMAX$ }
{ Post: Un segmento de línea recta ha sido dibujado en la pantalla entre los puntos (x1,y1) y (x2,y2) }
DibujarSegmento(entrada x1, y1, x2, y2 : real)

{ Pre: $|x| \leq XMAX \wedge |y| \leq YMAX \wedge 0 \leq r \wedge 0 \leq |x| + r \leq XMAX$
$\wedge\ 0 \leq |y| + r \leq YMAX$ }
{ Post: Un círculo con origen (x,y) y radio r , ha sido dibujado en la pantalla }

DibujarCírculo(entrada x, y, r : real)

Ejemplo de uso de la Máquina de Trazados:

Hacer un programa que dibuje un círculo y un cuadrado. El círculo tiene radio R. Suponga que R es positivo y menor que el mínimo valor entre XMAX y YMAX. Tanto el círculo como el cuadrado tienen el centro en el origen de coordenadas y el radio del círculo es igual a la mitad de la longitud de un lado del cuadrado.

Note que falta especificar aún más.... por ejemplo, un lado del cuadrado es paralelo al eje x.

El programa sería:

[**const** R: **real**;

{ $0 < R \leq \min(XMAX, YMAX)$ }

DibujarCírculo(0, 0, R);
DibujarCuadradoParaleloalEjeX(0, 0, 2*R)

{ Se ha dibujado en la pantalla un círculo de radio R y centro (0,0) $\wedge$ se ha dibujado un cuadrado con centro (0, 0), los lados de longitud 2*R y un lado del cuadrado es paralelo
al eje x }
]

proc DibujarCuadradoParaleloalEjeX(entrada x, y, long: real)
{ Pre: $|x| \leq XMAX \wedge |y| \leq YMAX \wedge 0 \leq long \wedge 0 \leq |x| + long/2 \leq XMAX$
$\wedge\ 0 \leq |y| + long/2 \leq YMAX$ }
{ Post: se ha dibujado en pantalla un cuadrado paralelo al eje horizontal. El centro del cuadrado es (x, y) y la longitud de los lados es long }
[
DibujarSegmento (x-long/2, y-long/2, x-long/2, y+long/2);
DibujarSegmento (x-long/2, y+long/2, x+long/2, y+long/2);
DibujarSegmento (x+long/2, y+long/2, x+long/2, y-long/2);
DibujarSegmento (x+long/2, y-long/2, x-long/2, y-long/2)
]

Note que para poder utilizar el procedimiento DibujarSegmento en el procedimiento DibujarCuadradoParaleloalEjeX, es necesario establecer la precondición dada. Y vemos que los argumentos de la llamada a DibujarCuadradoParaleloalEjeX satisfacen la precondición de este procedimiento. En caso de que la precondición de este procedimiento hubiese sido $|x| \leq XMAX \wedge |y| \leq YMAX \wedge 0 \leq long$, entonces corremos el riesgo de que el procedimiento no pueda dibujar el cuadrado, o simplemente se detenga la ejecución por haberse detectado un error.

Ejercicios:

1) demostrar la correctitud del procedimiento DibujarCuadradoParaleloalEjeX.
2) Especificar y hacer un programa que dibuje en pantalla 4 círculos concéntricos de centro (0, 0) y donde la distancia entre dos círculos consecutivos cualesquiera sea D (¿se entiende el término *distancia*?...). El círculo más pequeño tiene radio R.

4.3.2. Funciones

En el ejemplo anterior usamos un procedimiento para calcular el valor de una función. Sin embargo, resulta natural tener un constructor que se comporte como las funciones matemáticas, es decir, al hacer "una llamada" a la función con ciertos argumentos, la misma llamada represente el valor de la función para esos argumentos. Podemos incorporar el constructor función a nuestro pseudolenguaje. La declaración de una función en el pseudolenguaje es como sigue:

```
<tipo>  func  <identificador> (<especificación de parámetros>; ... ;
                               <especificación de parámetros>)
{ Pre: P}
{ Post: Q }
<cuerpo de la función>
```

La declaración es similar a la de un procedimiento. Las diferencias son:

- La palabra reservada **func** para denotar que se declara una función.
- <tipo> es el tipo o clase del valor resultante de la evaluación de la función (entero, real, etc.).
- Los parámetros formales deberán ser de **entrada**. Esta restricción es propia de nuestro pseudolenguaje.
- <cuerpo de función> es un programa que no incluye precondición ni postcondición. En una llamada a la función, la última instrucción que se ejecuta en <cuerpo de función> deberá ser siempre "devolver el valor resultado de la evaluación de la función". Esta instrucción se escribe en el pseudolenguaje de la siguiente forma:

  Devolver(<expresión>)

  donde, **Devolver** es una palabra reservada del pseudolenguaje, <expresión> representa una expresión cuyo tipo ó clase es <tipo>, y puede ser un identificador o una expresión (aritmética, lógica, etc.), por ejemplo: t*(s-1), x, etc. El valor resultante de evaluar <expresión> al momento de ejecutar la instrucción Devolver , será el valor de la función para los argumentos de entrada.

Un ejemplo de declaración de función sería:

real func F(entrada x: real)
{Pre: x=X }
{Post: devuelve el valor de la expresión $X^2 + 3X - 2$ }

```
[
 Devolver ( x*x + 3x – 2 )
]
```

<u>Definición formal de una llamada a función:</u>

En lo que sigue supondremos que una declaración de función tiene la forma siguiente:

```
<tipo>  func F(entrada x)
{ P }
{ Q }
  S
```

donde $\underline{x}$ representa una lista de los parámetros de entrada x_i. No tomamos en cuenta los tipos de los parámetros pues no intervienen en lo que expondremos a continuación.

Estamos interesados en definir formalmente la instrucción "llamada a una función" en el pseudolenguaje, la cual tiene la forma:

$$F(\underline{a})$$

El nombre de la función es F. $\underline{a}$ es la lista de los parámetros reales o argumentos a_i separados por comas. Los a_i son expresiones (por ejemplo, t ó t*s). Los a_i son los argumentos de entrada que corresponden respectivamente a los parámetros formales de entrada x_i. Cada argumento debe ser del mismo tipo que su parámetro formal correspondiente.

Los identificadores, sean nombres de variables o procedimientos, que están accesibles (puedan ser usados) al momento de la llamada a la función, deben ser diferentes de los parámetros formales $\underline{x}$ de la función. Esta restricción evita tener que usar notación extra para manejar el conflicto que se presenta al tener un mismo identificador utilizado con dos propósitos diferentes, pero no es esencial.

Una llamada a la función F del ejemplo dado antes es: F(a). Su ejecución calcula $a^2 + 3.a - 2$ y devuelve el valor resultante. F(a) representa finalmente ese valor resultante en el punto del programa donde se hace la llamada a F.

En general, la interpretación operacional de la llamada $F(\underline{a})$ es como sigue:

> Todos los parámetros son considerados variables locales de la función (su alcance es el cuerpo de la función). Primero, se determina los valores de los argumentos de **entrada** $\underline{a}$ y se almacenan en los parámetros correspondientes $\underline{x}$. Segundo, se ejecuta el cuerpo de la función. La última instrucción que se ejecuta en el cuerpo de la función deberá ser "Devolver(expresión)" donde "expresión", es una expresión del mismo tipo que aparece en la declaración de la función. El valor que representa $F(\underline{a})$ es el que resulta de evaluar esta expresión, y decimos que este valor es el que

“devuelve” F.

Una *llamada a función* deberá formar parte de una expresión en una instrucción de nuestro programa, por ejemplo:

$$z := (F(x1) – F(x2)) /(x1 – x2)$$

La instrucción de asignación anterior se interpreta operacionalmente de la siguiente manera: F(x1) y F(x2) representan los valores que “devuelve” la función F una vez que se ejecuta el cuerpo de la función para x1 y x2 respectivamente. Una vez calculados estos valores se procede a evaluar la expresión y a asignar a la variable z el valor resultante de esa evaluación.

De la interpretación operacional de una llamada a función, vemos que la ejecución de la instrucción c := F($\underline{a}$), donde interviene una llamada a la función F, es equivalente a la instrucción PF($\underline{a}$,c). Donde PF es un procedimiento con parámetros formales de entrada $\underline{x}$ (igual a los de F), un solo parámetro de salida *z*, del mismo tipo que la variable c, y el cuerpo del procedimiento PF, denotado por S_{PF}, igual al cuerpo de F reemplazando cada instrucción Devolver(<expresión>) por la instrucción *z* := <expresión>. La precondición P_{PF} de PF es la precondición P de F y la postcondición Q_{PF} de PF es la postcondición Q de F agregando que el valor de la función queda almacenado en *z*.

En nuestro ejemplo PF sería:

proc PF(entrada x: real; salida z: real)
{Pre: x=X }
{Post: $z = X^2 + 3X – 2$ }
[
  $z := x^2 + 3x – 2$
]

Por lo tanto, en general, demostrar que se cumple:

$$\{P1\}\ c := F(\underline{a})\ \{Q1\}$$

es equivalente a demostrar que se cumple:

$$\{P1\}\ PF(\underline{a}, c)\ \{Q1\}$$

Y esto último es equivalente a demostrar que se cumple:

$$\{ P1 \}\ \underline{x} := \underline{a}\ ;\ \{ P_{PF} \}\ S_{PF}\ ;\ \{ Q_{PF} \}\ c := z\ \{ Q1 \}$$

<u>Regla de llamada a función:</u>

Probar formalmente que se cumple:

$$\{ P1 \}\ c := F(\underline{a})\ \{ Q1 \}$$

Es equivalente a probar:

1) $P1 \Rightarrow P_{PF}(\underline{x} := \underline{a})$ es una tautología
2) $P1' \wedge Q_{PF}(\underline{x}, z := \underline{a}, c) \Rightarrow Q1$ es una tautología.

Donde P1 es de la forma $P1' \wedge R$ y P1' no depende de c.

Los parámetros de entrada $\underline{x}$ pueden aparecer en la postcondición Q. Sin embargo, es preferible utilizar, en la postcondición de una función, variables de especificación para denotar el valor inicial de los parámetros de entrada.

De igual forma, para demostrar que el cuerpo de una función cumple con las pre y post condiciones de la función basta con demostrar que el cuerpo de PF cumple con las pre y postcondiciones de PF.

Utilizando la función F, el programa que calcula (f(x1)-f(x2))/(x1-x2) es:

```
[ var x1, x2, z : real;
  { x1 = X1 ∧ x2 = X2 ∧ (X1 – X2)≠0 }
  z := (F(x1) – F(x2))/(x1 – x2)
  { z = (f(X1)-f(X2)) / (X1-X2), donde f es la función f(x)= x^2 + 3x - 2 }
]
```

vemos que este programa queda expresado completamente en términos de la información dada en el enunciado inicial, y por lo tanto es más fácil de entender.

Si queremos probar formalmente que se cumple:

$$\{ P2 \}\ z := (F(x1)-F(x2)) / (x1-x2)\ \{ Q2 \}$$

deberíamos probar que se cumple:

$$\{ P2 \}\ y1, y2 := F(x1), F(x2);\ z := (y1 - y2)/(x1 - x2)\ \{ Q2 \}$$

donde y1, y2 son variables nuevas que sólo utilizamos para la demostración de la correctitud de este trozo de programa.

Ejercicios:

1) Desarrolle una función que calcule el número de segundos correspondientes a una duración expresada en días, horas, minutos y segundos. Utilice dicha función para escribir un programa que dadas tres duraciones expresadas en días, horas, minutos y segundos, calcule la cantidad total de segundos de la suma de esas tres

duraciones.

4.4. Análisis por casos: La instrucción de selección (condicional o alternativa)

Cuando resolvemos un problema, el análisis por casos permite hacer una división del espacio de estados en subespacios y resolver el mismo problema para cada subespacio. La solución del problema original estará compuesta, de manera condicional, de las soluciones para cada subespacio. Dicho de otra forma, dado un estado inicial, la solución del problema está condicionada a ese estado inicial, pues se aplicará la solución que corresponda al subespacio que contiene a ese estado inicial.

Ejemplo:

Problema: Escribir un programa que determine el valor absoluto de un número real dado a.

Este problema se puede resolver particionando el espacio de estados en dos: $a < 0$ y $a \geq 0$. Si a es menor que 0 el resultado es –a, y en caso contrario el resultado será a.

Para particionar (o dividir) el espacio de estados contamos en el pseudolenguaje con la *instrucción de selección (o condicional o alternativa)*:

if $B_0 \rightarrow S_0$
[] $B_1 \rightarrow S_1$
[] ...
[] $B_n \rightarrow S_n$
fi

donde, **if** y **fi** son palabras reservadas del pseudolenguaje, B_i, $0 \leq i \leq n$, es una expresión booleana (su evaluación resulta en verdad o en falso), y S_i, $0 \leq i \leq n$, es una instrucción (recuerde que el operador de secuenciación es una instrucción, por lo que una secuencia de instrucciones es una instrucción). El constructor $B_i \rightarrow S_i$ se llama "*comando con guardia*" y la *guardia* es B_i. La interpretación operacional de la instrucción de selección es la siguiente:

En la ejecución de una instrucción de selección todas las guardias son evaluadas. Si ninguna de las guardias es verdadera entonces la ejecución de la instrucción de selección produce error, en caso contrario, se escoge una de las guardias con valor "verdad" y se ejecuta la secuencia de instrucciones correspondiente. Note que no especificamos cual guardia escogemos de las que resultan "verdad", podría ser la primera de arriba hacia abajo. Pero preferimos dejar sin especificar este hecho (*lo dejamos indeterminado*).

Ejemplos de instrucciones de selección válidas en nuestro formalismo:

```
if (a ≥ 3) → x := 4 []  (a ≤ 3) →  x := 5  fi

if (a ≥ 3) → x := 4; b := a*2   []  (a < 3) →  a := 5  fi

if (a ≥ 3) → x := 4; b := a*2   []
    (a < 3) →  if (x ≥ 3) → x := 4 []  (x < 3) →  x := 5  fi; a := 5
fi

if ((x ≥ 3)∧p) → x := 4 []  (¬p) →  skip  fi
```

Note que si antes de ejecutarse la última instrucción el valor de las variables es x=2 y p=V entonces al ejecutarse la instrucción dará error y el programa, donde está inmersa esta instrucción, no podrá continuar ejecutándose.

La solución tentativa al problema de cálculo del valor absoluto de un número real es:

```
[ const A: real;
   var b: real;
  { verdad }
  if  (A < 0) → b := -A
  []  (A ≥ 0) → b := A
  fi
  { b = |A| }
]
```

Nota: **verdad** y **falso** son palabras reservadas del pseudolenguaje para denotar los valores de verdad.

Regla de la instrucción de selección:

Ahora definimos la regla que permite demostrar la correctitud de una instrucción de selección, lo haremos para dos guardias:

Probar que se cumple: { P } **if** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **fi** { Q }

es equivalente a probar que se cumple:

1) $P \Rightarrow$ B_0 y B_1 están bien definidas, es una tautología.
2) $P \Rightarrow B_0 \vee B_1$, es una tautología .
3) $\{ P \wedge B_0 \} S_0 \{ Q \}$ se cumple.
4) $\{ P \wedge B_1 \} S_1 \{ Q \}$ se cumple.

Que B_i esté bien definida si P es verdad, significa que la expresión se pueda evaluar para los estados que cumplan P; por ejemplo, que no haya división por cero. Normalmente omitimos demostrar (1) por ser evidente.

Note que en la ejecución del **if**, independientemente de cual comando con guardia es el que se ejecuta, al final de la ejecución se debe cumplir Q. Lo importante es que la ejecución de cualquier instrucción donde la guardia sea verdadera lleva a un resultado correcto, el programador no tiene que preocuparse por cuál será ejecutado. Por lo tanto, el programador es libre de programar tantos comandos con guardia como desee sin tomar en cuenta que más de una guardia pueda ser verdad en el mismo momento.

Demostremos la correctitud del programa dado antes para calcular el valor absoluto:

1) $P \Rightarrow B_0$ y B_1 están bien definidas

"verdad $\Rightarrow (A < 0)$ y $(A \geq 0)$ están bien definidas" es equivalente a "$(A < 0)$ y $(A \geq 0)$ están bien definidas". Y en efecto, esas dos expresiones están bien definidas porque A es un número real.

2) $P \Rightarrow B_0 \vee B_1$, es una tautología

De nuevo, "verdad $\Rightarrow (A < 0) \vee (A \geq 0)$" es equivalente a "$(A < 0) \vee (A \geq 0)$", y esta última expresión siempre es verdad cualquiera sea el número real A.

3) $\{ P \wedge B_0 \}\ S_0\ \{ Q \}$ se cumple

¿$\{ \text{verdad} \wedge (A < 0) \}\ b := -A\ \{ b = |A| \}$ se cumple?:

$(b = |A|)(b := -A)$
≡ por sustitución
$(-A = |A|)$
≡ por aritmética
$(A \leq 0)$
⇐ por aritmética
$(A < 0)$
≡ por cálculo de predicados
$\text{verdad} \wedge (A < 0)$

4) $\{ P \wedge B_1 \}\ S_1\ \{ Q \}$ se cumple

¿$\{ \text{verdad} \wedge (A \geq 0) \}\ b := A\ \{ b = |A| \}$ se cumple?:

$(b = |A|)(b := A)$
≡ por sustitución
$(A = |A|)$
≡ por aritmética
$(A \geq 0)$
≡ por cálculo de predicados
$\text{verdad} \wedge (A \geq 0)$

Utilizaremos las siguientes anotaciones en los programas, indicando además, las demostraciones correspondientes:

{ P }
if $B_0 \rightarrow \{ P \wedge B_0 \}$ S_0 { Q, Demostración 0 }
[] $B_1 \rightarrow \{ P \wedge B_1 \}$ S_1 { Q, Demostración 1 }
fi
{ Q, Demostración 2 }

con:
Demostración 0: la demostración de que $\{ P \wedge B_0 \}$ S_0 { Q } se cumple.
Demostración 1: la demostración de que $\{ P \wedge B_1 \}$ S_1 { Q } se cumple.
Demostración 2: la demostración de que $P \Rightarrow B_0 \vee B_1$, es una tautología y, si es relevante, una demostración de que $P \Rightarrow$ B_0 y B_1 están bien definidas, es una tautología.

Ejercicios:

1) Probar que se cumple:
   a) {x = 0} if verdad $\rightarrow$ x := x+1 [] verdad $\rightarrow$ x := x+1 fi { x = 1}
   b) { x = 0 } if verdad $\rightarrow$ x := 1 [] verdad $\rightarrow$ x := -1 fi { $x = 1 \vee x = -1$ }
2) Ejercicios 0, 1, 2 de la página 27 del Kaldewaij
3) Hacer un programa correcto que resuelva los problemas 2, 3, 5, 6 de la sección 1 del problemario de Michel Cunto.
4) Hacer ejercicios 1, 3, 4, 5, 6, sección 1 del problemario de Michel Cunto.

Ejemplo de síntesis (uso de procedimientos, funciones, instrucción condicional, diseño descendente):

Problema: Hacer un procedimiento que calcule todas soluciones reales de la ecuación $Ax^2+Bx+C=0$ con coeficientes reales A, B, C. Suponga que cuenta con una función que permite calcular la raíz cuadrada de un número real no negativo. La especificación de esta función es como sigue:

{ Pre: $x=X \wedge X \geq 0$ }
{ Post: devuelve $\sqrt{X}$ }
real RaizCuadrada(**entrada** x : **real**)

Una raíz real de un polinomio de segundo grado $A.x^2 + B.x + C$ es un número real r ($r \in \Re$) tal que $A.r^2 + B.r + C=0$. En la especificación que haremos, la variable "conj" representa el conjunto de las raíces reales del polinomio $A.x^2 + B.x + C$.

La especificación del programa sería:

[const A, B, C: real;
  var conj: conjunto de números complejos;

{ Pre: verdad }
S
{Post: $conj = \{x : (x \in \Re) \wedge (A.x^2 + B.x + C = 0)\}$ }
]

Como no se impone ninguna condición a los valores de los coeficientes del polinomio, debemos hacer un análisis por casos; es decir, dividimos el espacio de estados y dependiendo del valor que puedan tener los coeficientes habrá una solución distinta al problema.

Por la *teoría asociada* a la especificación del problema, el polinomio será de segundo grado cuando $a \neq 0$ y las raíces vienen dadas por la ecuación:

$$\frac{-B \pm \sqrt{B^2 - 4.A.C}}{2.A}$$

tendrá dos raíces reales si el discriminante, $B^2 - 4.A.C$, es mayor que cero. No tendrá raíces reales si el discriminante es negativo. Tendrá una sola raíz real si el discriminante es cero. Cuando $A = 0$ y $B \neq 0$, el polinomio es de primer grado, y existirá una sola raíz real. Y cuando $A = 0$, $B = 0$ y $C = 0$, todo número real es solución. Cuando $A=0$, $B=0$ y $C \neq 0$, no existirá solución.

Por lo tanto podemos hacer una especificación más concreta (refinar la especificación) donde introducimos una variable entera n que nos indica el número de soluciones que tiene la ecuación $A.x^2 + B.x + C=0$. Si no hay solución entonces n=0. Si tiene una sola raíz, entonces n=1 y la variable x1 contendrá la raíz. Si hay dos soluciones entonces n=2 y las soluciones quedarán almacenadas en x1 y x2. Si todo número real es solución entonces n=3. Una especificación más concreta (o refinada) sería:

[const A, B, C: real;
var x1, x2: real;
var n: entero;
{ Pre: verdad }
S
{Post: $(n=0 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C \neq 0))$ $\vee$
$(n=1 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0 \equiv x = x1))$ $\vee$
$(n=2 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0 \equiv x = x1 \vee x = x2))$ $\vee$
$(n=3 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0))$ }
]

Sean Q0, Q1, Q2 y Q3 respectivamente los predicados:

$(n=0 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C \neq 0))$
$(n=1 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0 \equiv x = x1))$
$(n=2 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0 \equiv x = x1 \vee x = x2))$

$(n=3 \wedge (\forall x: x \in \Re: A.x^2 + B.x + C = 0))$

Note la forma en que se escribió la existencia de una o dos soluciones únicas de la ecuación, en los predicados Q1 y Q2. En el lenguaje de la lógica, cuando queremos expresar que un objeto d satisface una propiedad P y es el único que la satisface, podemos escribir:

$$P(d) \wedge (\forall x: : x \neq d \Rightarrow \neg P(x))$$

Sin embargo, esto se puede expresar de una manera más concisa como sigue:

$$(1) \qquad (\forall x: : P(x) \equiv x=d)$$

Veamos que esta última fórmula es equivalente a la primera:

$(\forall x: : P(x) \equiv x=d)$
$\equiv$ traducción de la equivalencia
$(\forall x: : (P(x) \Rightarrow x=d) \wedge (x=d \Rightarrow P(x)))$
$\equiv$ distribución de $\forall$ sobre $\wedge$
$(\forall x: : P(x) \Rightarrow x=d) \wedge (\forall x: : x=d \Rightarrow P(x))$
$\equiv$ simplificación de la segunda expresión
$(\forall x: : P(x) \Rightarrow x=d) \wedge P(d)$
$\equiv$ contrarecíproco en la primera expresión
$(\forall x: : x \neq d \Rightarrow \neg P(x)) \wedge P(d)$
$\equiv$ conmutatividad de $\wedge$
$P(d) \wedge (\forall x: : x \neq d \Rightarrow \neg P(x))$

Igualmente, expresar que dos objetos d y e son los únicos que satisfacen P puede formalizarse como:

$$P(d) \wedge P(e) \wedge (\forall x: : x \neq d \wedge x \neq e \Rightarrow \neg P(x))$$

O, de manera más concisa, pero equivalente, como:

$$(2) \qquad (\forall x: : P(x) \equiv x=d \vee x=e)$$

La demostración entre estas dos fórmulas es análoga a la anterior, aunque utiliza algunas reglas lógicas adicionales.

Los esquemas de modelación (1) y (2) fueron utilizados en la formulación de Q1 y Q2.

Veamos ahora, a efectos de desarrollar el programa, cómo puede ser reescrita la postcondición $Q0 \vee Q1 \vee Q2 \vee Q3$ en términos de los datos de entrada A, B y C, con la finalidad de obtener una postcondición que nos permita desarrollar un programa.

Veamos informalmente, utilizando la teoría asociada a la especificación, la relación que existe entre los valores de A, B y C y la existencia de 0, 1, 2 o más soluciones de la ecuación $A.x^2 + B.x + C = 0$.

Tenemos que la ecuación en cuestión no tiene solución en los números reales si A y B son cero y C no lo es, o cuando $A \neq 0$ y $B^2 - 4.A.C < 0$. Y este es el único caso en que no hay solución, lo cual indica que $(A=0 \wedge B=0 \wedge C\neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0)$ no es sólo condición suficiente sino además condición necesaria para que no exista solución. Formalmente tenemos:

$$(A=0 \wedge B=0 \wedge C\neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0) \equiv (\forall x: x \in \Re: A.x^2 + B.x + C \neq 0)$$

Lo que nos permite transformar el primer caso Q0 de nuestra postcondición en:

$$Q0': (n = 0 \wedge ((A=0 \wedge B=0 \wedge C\neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0)))$$

La existencia de una única solución se da cuando el polinomio tiene grado 1, esto es, cuando A es cero y B no lo es, o cuando el polinomio tiene grado 2 y ambas soluciones coinciden. Esto último ocurre cuando A no es cero y el discriminante es cero. Tenemos entonces lo siguiente:

$$(A = 0 \wedge B \neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C = 0) \equiv$$
$$(\exists sol : sol \in \Re : (\forall x: x \in \Re: A.x^2 + B.x + C = 0 \equiv x = sol))$$

La transformación de Q1 no es tan directa como la de Q0, pero nos podemos valer del siguiente hecho (demuéstrelo):

> Si tenemos que la existencia de un único elemento que satisface la propiedad P es equivalente a cierta condición R, esto es:
>
> $R \equiv (\exists sol : : (\forall x: : P(x) \equiv x = sol))$
>
> Entonces, el hecho de que un cierto elemento d sea el único que satisfaga P equivale a que se cumpla R y d satisfaga P, esto es:
>
> $(\forall x: : P(x) \equiv x = d) \equiv R \wedge P(d)$

Si tomamos a:

$((A = 0 \wedge B \neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C = 0))$ como R
$A.x^2 + B.x + C = 0$ como $P(x)$
x1 como d

y aplicamos el resultado anterior, podemos transformar el segundo caso Q1 de nuestra postcondición al predicado equivalente siguiente:

$$Q1': (n = 1 \wedge ((A = 0 \wedge B \neq 0) \vee (A \neq 0 \wedge B^2 - 4.A.C = 0))$$

$$\wedge \ (A.x1^2 + B.x1 + C = 0) \ )$$

Para la transformación de Q2, nuestra teoría sobre existencia de raíces reales del polinomio indica que:

$(A \neq 0 \ \wedge \ B^2 - 4.A.C > 0) \ \equiv$
$(\exists \text{ sol1, sol2} : \ : \text{sol1} \neq \text{sol2} \wedge (\ \forall x : \ : \ A.x^2 + B.x + C = 0 \ \equiv \ x = \text{sol1} \vee \ x = \text{sol2}$
$)$

Y podemos utilizar una propiedad similar a la anterior:

```
Si tenemos que:

R ≡(∃ sol1,sol2:: sol1 ≠ sol2 ∧ ( ∀x:: P(x) ≡ x = sol1 ∨ x = sol2 )
)

Entonces se cumple que:

(  ∀x: : P(x) ≡ x = d ∨  x = e ) ∧ d ≠ e ≡  R ∧ P(d) ∧ P(e) ∧ d ≠ e
```

Obtenemos entonces que Q2 es equivalente a:

$$Q2': \ (n = 2 \wedge A \neq 0 \ \wedge \ B^2 - 4.A.C > 0 \ \wedge \ (A.x1^2 + B.x1 + C = 0) \\ \wedge \ (A.x2^2 + B.x2 + C = 0) \wedge \ x1 \neq x2 \ )$$

Finalmente, la teoría relevante al último caso nos dice que:

$$A = 0 \wedge B = 0 \wedge C = 0 \ \equiv \ (\ \forall x : x \in \Re : \ A.x^2 + B.x + C = 0)$$

Por lo que el último caso, Q3, de la postcondición se puede escribir como:

$$Q3': (n = 3 \ \wedge \ A = 0 \wedge B = 0 \wedge C = 0)$$

Como resultado total, hemos reescrito la postcondición:

$$Q0 \vee Q1 \vee Q2 \vee Q3$$

Como:

$$Q0' \vee Q1' \vee Q2' \vee Q3'$$

Que nos da más información sobre cómo discriminar los cuatro casos en función de los valores de A, B y C, es decir, de los datos de entrada.

La nueva especificación sería:

[const A, B, C: real;

```
   var x1, x2: real;
   var n: entero;
   { Pre: verdad }
   S
   {Post: ( n = 0  ∧  ( (A=0 ∧ B=0 ∧ C≠0)  ∨  ( A ≠ 0 ∧ B² – 4.A.C < 0 ) ) )          ∨
          ( n = 1  ∧  ( ( A = 0  ∧  B ≠ 0 )  ∨  (  A ≠ 0  ∧  B²- 4.A.C = 0 ) )
                                          ∧  ( A.x1² + B.x1 + C = 0 ) )               ∨
          ( n = 2  ∧  A ≠ 0  ∧  B²- 4.A.C > 0  ∧   ( A.x1² + B.x1 + C = 0 )
                                          ∧  ( A.x2² + B.x2 + C = 0 ) ∧   x1 ≠ x2  )   ∨
          ( n = 3  ∧  A = 0 ∧ B = 0 ∧ C = 0 )   }
]
```

Note que los cuatro casos de la nueva postcondición cubren el espacio de estados, es decir, están considerados todos los estados posibles del espacio de estados. Una forma de comprobar esto es haciendo una tabla de decisión. Sean:

C00: $(A = 0 \wedge B = 0 \wedge C \neq 0)$
C01: $(A \neq 0 \wedge B^2 - 4.A.C < 0)$
C10: $(A = 0 \wedge B \neq 0)$
C11: $(A \neq 0 \wedge B^2 - 4.A.C = 0)$
C2: $(A \neq 0 \wedge B^2 - 4.A.C > 0)$
C3: $(A = 0 \wedge B = 0 \wedge C = 0)$

	A	B	C	Discriminante
C00	0	0	<	-
C00	0	0	>	-
C01	<	-	-	<
C01	>	-	-	<
C10	0	<	-	-
C10	0	>	-	-
C11	<	-	-	0
C11	>	-	-	0
C2	<	-	-	>
C2	>	-	-	>
C3	0	0	0	-

Donde < significa menor que cero, > significa mayor que cero, 0 significa igual a cero, - significa que puede ser cualquier valor. Con la tabla anterior podemos comprobar que cualquier combinación de valores <, >, 0 de A, B, C y Discriminante, está cubierta por una de las 6 condiciones C00, C01, C10, C11, C2 ó C3.

Un primer programa en el proceso de refinamiento sucesivo hasta encontrar una solución del problema en términos de los constructores del pseudolenguaje (es decir, hasta encontrar un programa) vendrá dado por una instrucción de selección, que se infiere directamente de la postcondición: Este primer refinamiento no diferencia los casos C01, C11 y C2

```
[  const A, B, C: real;
   var x1, x2: real;
   var n: entero;
   { Pre: verdad }
    if
          (A = 0  ∧ B = 0 ∧ C = 0) → S3
    []    (A = 0  ∧ B = 0 ∧ C ≠ 0) → S00
    []    (A = 0  ∧ B ≠ 0 ) → S10
    []    (A ≠ 0) → S01y11y2
    fi
   {Post: Q0' ∨ Q1' ∨ Q2' ∨ Q3'   }
]
```

Ahora pasamos a desarrollar S3, S00, S10 de forma que se cumpla lo siguiente:

$\{ Pre \wedge A = 0 \wedge B = 0 \wedge C = 0 \}$ S3 $\{ Post \}$
$\{ Pre \wedge A = 0 \wedge B = 0 \wedge C \neq 0 \}$ S00 $\{ Post \}$
$\{ Pre \wedge A = 0 \wedge B \neq 0 \}$ S10 $\{ Post \}$
$\{ Pre \wedge A \neq 0 \}$ S01y11y2 $\{ Post \}$

S3 sería:

```
n := 3
```

S00 sería:

```
n := 0
```

S10 sería:

```
n := 1;
x1 := -C/B
```

Ejercicio: Demuestre que las instrucciones dadas S3, S00 Y S10 cumplen la especificación anterior.

S01y11y2 es el trozo de programa más complicado a desarrollar, pues debemos determinar si un polinomio de segundo grado tiene dos raíces reales, o una sola raíz o ninguna. Las ecuaciones que representan los valores de las raíces x1 y x2 de un polinomio de segundo grado $A.x^2 + B.x + C$ en función de los coeficientes vienen dadas por la expresión:

$$\frac{-B \pm \sqrt{B^2 - 4.A.C}}{2.A}$$

Ahora bien, dependiendo de si el discriminante (B^2-4.A.C) es cero, positivo o negativo, tendremos una sola raíz, dos raíces o ninguna raíz real, respectivamente. las raíces serán

reales o complejas. Otra vez nos encontramos con un análisis por casos.

Si (B^2-4.A.C) es cero, tenemos una sola ráiz real x1 = -B/(2.A)

Si (B^2-4.A.C) es positivo, las raíces serán $x1 = \frac{-B + \sqrt{B^2 - 4.A.C}}{2.A}$ y $x2 = \frac{-B - \sqrt{B^2 - 4.A.C}}{2.A}$

Si (B^2-4.A.C) es negativo, no existen raíces reales.

Por lo tanto S01y11y2 sería:

```
disc := B*B-4*A*C;
if  disc = 0 →   n := 1; x1 := -B/(2*A)
[]  disc > 0 →   n := 2;
                 x1 := (-B + RaizCuadrada(-disc))/(2*A);
                 x2 := (-B - RaizCuadrada(-disc))/(2*A);
[]  disc < 0 →   n := 0
fi
```

Note que hemos introducido una nueva variable, disc, para mejorar la eficiencia del algoritmo en cuanto a tiempo. Se calcula una sola vez la expresión (B*B-4*A*C) y se utiliza 5 veces.

Si convertimos el programa completo en un procedimiento este sería:

```
proc RaicesPolinomio( entrada A, B, C: real; salida n, x1, x2: real)
{Pre: verdad }
{Post:  ( n = 0 ∧ ( (A=0 ∧ B=0 ∧ C≠0) ∨ ( A ≠ 0 ∧ B² – 4.A.C < 0 ) ) )          ∨
        ( n = 1 ∧ ( ( A = 0 ∧ B ≠ 0 ) ∨ ( A ≠ 0 ∧ B²- 4.A.C = 0 ) )
                                   ∧ ( A.x1² + B.x1 + C = 0 ) )                  ∨
        ( n = 2 ∧ A ≠ 0 ∧ B²- 4.A.C > 0 ∧ ( A.x1² + B.x1 + C = 0 )
                                   ∧ ( A.x2² + B.x2 + C = 0 ) ∧ x1 ≠ x2 )        ∨
        ( n = 3 ∧ A = 0 ∧ B = 0 ∧ C = 0 ) }
[ var disc: real;
  if
       (A = 0 ∧ B = 0 ∧ C = 0) → n := 3
  []   (A = 0 ∧ B = 0 ∧ C ≠ 0) → n := 0
  []   (A = 0 ∧ B ≠ 0 ) →  n := 1;
                           x1 := -C/B
  []   (A ≠ 0) →       disc := B*B-4*A*C;
                       if disc = 0 → n := 1; x1 := -B/(2*A)
                       [] disc > 0 → n := 2;
```

```
                        x1 := (-B + RaizCuadrada(-disc))/(2*A);
                        x2 := (-B - RaizCuadrada(-disc))/(2*A);
              []  disc < 0 → n := 0
              fi

   fi
]
```

Ejercicios:

1) Demuestre que se cumple: { Pre $\wedge$ A $\neq$ 0 } S01y11y2 { Post }
2) Complete la demostración de correctitud del programa anterior.

4.5. Análisis de Procesos Iterativos: La instrucción iterativa

Hasta ahora podemos describir un número muy reducido de procesos con los constructores del pseudolenguaje: aquéllos donde el número de acciones que se ejecutan para obtener el resultado es proporcional al número de constructores utilizados en el programa que describe dicho proceso. Hasta el momento no podemos describir con nuestro pseudolenguaje el proceso de pelar papas visto en el capítulo 1. Necesitamos un constructor, que llamaremos *instrucción iterativa ó instrucción de repetición*, que nos permita describir un *proceso iterativo*, es decir, un proceso que consiste en la ejecución de una misma acción A un número dado de veces. El número de veces que se ejecuta la acción A depende del estado de las variables antes de ejecutarse la acción. En lenguaje natural decíamos: "Mientras haya papas en el cesto, pelar una papa". Esta descripción nos dice que se debe efectuar la siguiente acción repetidas veces: verificar si el cesto tiene papas y de ser así se debe pelar una papa. Si el cesto no tiene papas, el proceso termina. La *condición de continuación* del proceso iterativo es "el cesto tiene papas". Por supuesto, la acción que se repite debe ser capaz de modificar el estado de las variables con el fin de garantizar que el proceso descrito termine. En un proceso iterativo, la acción que consiste en "verificar la condición de continuación y luego, dependiendo del resultado de la evaluación, ejecutar la acción respectiva" la denominamos *una iteración*. Note que la última iteración corresponde a que no se cumple la condición de continuación y es cuando termina el proceso iterativo. Algunos autores no consideran este último paso del proceso iterativo como una iteración, sin embargo, el análisis que haremos se facilitará si lo consideramos una iteración.

El número de iteraciones que realiza una instrucción iterativa es variable y depende del estado inicial de las variables al momento de ejecutarse la instrucción iterativa, de la condición de continuación y de los cambios de estado producidos por la acción que se repite. Por lo tanto, una instrucción iterativa será una descripción muy concisa de un proceso que puede efectuar un número considerablemente grande de acciones.

Una instrucción iterativa es usada típicamente para describir el proceso que consiste en recorrer, uno a uno, los objetos de algún espacio (por ejemplo, un conjunto de números, una secuencia de objetos, cesto con papas, etc.) y efectuar la misma acción a cada objeto del espacio (pelar la papa, sumar el valor del objeto a una variable que mantiene un

acumulado, etc.). Se examinan los objetos del espacio para:

1) Buscar alguno (o todos) que cumpla ciertas propiedades. Por ejemplo, si queremos buscar el menor factor primo de un número entero mayor que 1, **x,** podríamos proceder recorriendo (de menor a mayor) los números naturales entre 2 y **x** hasta encontrar un número primo que divida a **x**.

2) Aplicar una función a cada objeto. Por ejemplo, aumentar un 10% el sueldo de cada empleado.

3) Combinar los objetos de alguna forma. Por ejemplo: sumar los cuadrados de los primeros n números naturales.

4) Aplicar combinaciones de los anteriores. Por ejemplo, aumentar los sueldos y reportar la suma total de los nuevos sueldos.

Una forma de capturar la esencia de la acción global que realiza un proceso iterativo (es decir, en qué consiste el proceso iterativo) es determinando una aserción que refleje el estado del proceso al comienzo de cualquier iteración del mismo, incluso al comienzo de la iteración en la que no se satisface la condición de continuación. A este tipo de aserciones la llamamos *un invariante* del proceso iterativo (o de la instrucción iterativa que describe al proceso), porque no varía de una iteración a otra.

Por ejemplo, una manera de calcular la suma de los cuadrados de los primeros N (N≥0) números naturales comenzando desde 0 (es decir, la suma $0^2 + 1^2 + ... + (N-1)^2$) es efectuando el proceso iterativo que consiste en recorrer los números naturales desde 0 hasta N y, para cada número natural que obtengamos en el recorrido, sumamos su cuadrado a una variable, llamémosla **suma**, que inicializamos en cero; cuando obtenemos el número natural N concluímos el proceso iterativo sin sumar su cuadrado a **suma**. La acción que se realiza en la iteración i es "sumar i^2 a **suma** e incrementar i en 1" (en este caso el número de la iteración, comenzando con la iteración cero, coincide con el número natural cuyo cuadrado se acumula en esa iteración). La condición de continuación del proceso iterativo es $i < N$. Por lo tanto la esencia del proceso la podemos capturar por la aserción (o predicado) siguiente: "al comienzo de la iteración i, **suma** contiene la suma de los cuadrados de los primeros números naturales entre 0 y (i-1) y $0 \leq i \leq N$". Si originalmente N es igual a cero, vemos que **suma** contendrá cero, lo cual es consistente (la suma de cero números es cero). Por supuesto, la condición de terminación del proceso iterativo es la negación de la condición de continuación, y en nuestro ejemplo, la condición de terminación es "al comienzo de la iteración i el número natural cuyo cuadrado sumaremos a **suma** en esa iteración es mayor o igual a N". Entonces, vemos que una vez se cumpla la condición de terminación, es decir, $i \geq N$, se seguirá cumpliendo el invariante y tendremos $N \leq i \leq N$, por lo que $i = N$ y **suma** contendrá la suma de los cuadrados de los números naturales entre 0 y N-1.

Al predicado "al comienzo de la iteración i, **suma** contiene la suma de los cuadrados de los primeros números naturales entre 0 y (i-1) y $0 \leq i \leq N$" lo llamamos *un invariante* del

proceso iterativo. Se usa la palabra invariante por el hecho de que es un predicado que se cumple, es una aserción, al comienzo de una iteración cualquiera del proceso iterativo. Note que un proceso iterativo puede tener muchos invariantes, por ejemplo "verdad" es un invariante (si suponemos que las operaciones que realiza el proceso están bien definidas, al comienzo de cada iteración, siempre se cumplirá el predicado "verdad"). Sin embargo, estamos interesados principalmente en conseguir un invariante que capture la esencia del proceso iterativo.

Ejercicios:

1) Determine un proceso iterativo y un invariante que capture el proceso, para calcular $(\sum i: 0 \leq i < n : s[i])$. Donde s es una secuencia de largo n.
2) Determine un proceso iterativo y un invariante que capture el proceso, para calcular $(\prod i: 0 \leq i < n : 1/(i+1))$.
3) Determine un proceso iterativo y un invariante que capture el proceso, para calcular el mayor elemento de una secuencia s de números enteros de largo n.
4) Determine un proceso iterativo y un invariante que capture el proceso, para ordenar de menor a mayor los elementos de una secuencia s de números enteros de largo n (la acción repetitiva no tiene por qué comentarla con precisión)
5) Determine un proceso iterativo y un invariante que capture el proceso, de los ejercicios 12, 13, 15, 16, 17, 24, 25, 26, 30 de la sección 1 del problemario de Michel Cunto.

La instrucción iterativa tiene la forma siguiente:

do $B_0 \rightarrow S_0$
[] $B_1 \rightarrow S_1$
[] ...
[] $B_n \rightarrow S_n$
od

donde, **do** y **od** son palabras reservadas del pseudolenguaje, B_i, $0 \leq i \leq n$, es una expresión booleana (es una guardia, su evaluación resulta en verdad o en falso), y S_i, $0 \leq i \leq n$, es una instrucción. Cada $B_i \rightarrow S_i$ es un comando con guardia, cuya guardia es B_i. La interpretación operacional de la instrucción iterativa es la siguiente:

> Se evalúan todas las guardias. Si todas las guardias son "falso" (esto equivale a la condición de terminación), entonces se ejecuta la instrucción **skip**. En caso contrario, se escoge una guardia con valor "verdad" y se procede a ejecutar la instrucción correspondiente a esa guardia; una vez ejecutada la instrucción, se procede a ejecutar la instrucción iterativa nuevamente.

Cada iteración corresponderá a la ejecución del *cuerpo de la instrucción iterativa (la acción que se repite),* que es:

$B_0 \rightarrow S_0$

```
[]      B1 → S1
[]       ...
[]      Bn → Sn
```

Ahora podemos escribir un programa que describa el proceso iterativo visto más arriba, para el cálculo de la suma de los cuadrados de los números naturales de 0 a N, en términos de la instrucción iterativa del pseudolenguaje:

```
                 [  const N: entero;
                    var i, suma: entero;
                    { N ≥ 0 }

                    suma := 0;
        (1)         i := 0;
                    do
                          i < N  → suma := suma + i*i; i := i+1
                    od

                    { suma = ( ∑i: 0 ≤ i < N : i^2 )  }
                 ]
```

En cualquier instante de la ejecución del programa, al comienzo del cuerpo de la iteración, el estado de la variable i corresponde al número de la iteración que será ejecutada a partir de ese instante (comenzando con la iteración 0). Esa misma variable i permite “recorrer” los números naturales de 0 a N. Por lo tanto, el predicado: “la variable i corresponde al número de la iteración que será ejecutada a partir de ese instante”, es un invariante del proceso iterativo descrito anteriormente (también decimos que es un *invariante de la instrucción iterativa*). Sin embargo, este invariante no captura la esencia del proceso que se lleva a cabo. El invariante que captura esa esencia es el siguiente: “**suma** $= (\sum j: 0 \leq j < i : j^2) \wedge 0 \leq i \leq N$”. Más adelante aprenderemos a demostrar formalmente que en efecto este es un invariante de la instrucción iterativa anterior; por los momentos sólo estamos suponiendo que este es un invariante.

Note que justo antes de la instrucción **do**... se cumple este invariante, si sustituimos a **suma** y a **i** por 0, es decir, por sus valores originales: “ **0** $= (\sum j: 0 \leq j < 0 : j^2) \wedge 0 \leq 0 \leq N$”, lo cual es verdad. También se deberá cumplir cuando la guardia dé valor falso, es decir, cuando $i \geq N$ (condición de terminación): “**suma** $= (\sum j: 0 \leq j < i : j^2) \wedge 0 \leq i \leq N$”; y vemos entonces que al concluir la ejecución de la instrucción iterativa, la variable suma contendrá efectivamente el valor esperado, es decir, la suma de los cuadrados de los primeros N números naturales, es este hecho el que nos permite decir que este invariante “captura la esencia del proceso iterativo”. Esto muestra la importancia de encontrar un invariante para demostrar que una instrucción iterativa satisface una especificación dada.

Hay que demostrar también que la instrucción iterativa terminará en algún momento, es decir, que *no caerá en un ciclo infinito*. Esto lo podemos garantizar, en el ejemplo anterior, sabiendo que la variable i aumenta estrictamente en cada iteración y que la

guardia dice que se parará el proceso iterativo una vez i alcance el valor N. Por lo tanto hay dos cosas que deben ser probadas para demostrar que una instrucción iterativa satisface una especificación y son: encontrar el invariante que captura el proceso, lo cual permite demostrar que la postcondición se cumplirá al final, y garantizar que la ejecución de la iteración termina.

La instrucción iterativa es el último constructor que daremos del pseudolenguaje. Es claro que es el más complejo. En efecto, éste es la esencia de la programación *imperativa* o *secuencial*. Es por esto que se puede decir que la actividad de la programación consiste principalmente en tener destreza para determinar invariantes. La ciencia de la computación nos dice que los pocos constructores que hemos dado son suficientes para resolver *cualquier* problema que admita una solución algorítmica!!

<u>Definición formal de una instrucción iterativa</u>:

Para demostrar formalmente la correctitud de una instrucción iterativa utilizaremos una regla que llamaremos "Teorema de la Invariancia". Esta regla depende fuertemente del concepto de invariante que vimos antes.

Consideremos que la instrucción iterativa posee una sola guardia. De la interpretación operacional de la instrucción iterativa podemos concluir lo siguiente:

Demostrar que se cumple { P } **do** B → S **od** { Q }, es equivalente a demostrar que se cumple:

{ P }
if ¬B → **skip**
[] B → S; **do** B → S **od**
fi
{ Q }

Haciendo las anotaciones en la instrucción condicional, obtenemos:

{ P }
if ¬B → { P ∧ ¬B} **skip** { Q }
[] B → { P ∧ B} S; **do** B → S **od** { Q }
fi
{ Q }

Como { P } **do** B → S **od** { Q } debería cumplirse, escogemos P como predicado intermedio en la secuenciación "S; **do** B → S **od"**. Note que un predicado P que cumpla con lo anterior es un invariante. Obtenemos:

{ P }
if ¬B → { P ∧ ¬B} **skip** { Q }
[] B → { P ∧ B} S { P }; **do** B → S **od** { Q }

fi
{ Q }

Así, habría que demostrar:

(i) $[P \wedge \neg B \Rightarrow Q]$ (es decir, el predicado es una tautología), que surge de la regla de correctitud de la instrucción **skip**.
(ii) $\{ P \wedge B\}$ S $\{ P \}$ se cumple
(iii) { P } **do** $B \rightarrow S$ **od** { Q }

en donde (iii) correspondería de nuevo a (i), (ii) y (iii). Si podemos asegurar que la instrucción iterativa termina, entonces será suficiente probar (i) y (ii).

Lo anterior lo podemos formular para dos guardias, suponiendo terminación, como sigue.

Regla de la instrucción iterativa:

Si
(i) $[P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q]$ y
(ii) $\{ P \wedge B_0\}$ S_0 $\{ P \}$ y $\{ P \wedge B_1\}$ S_1 $\{ P \}$ se cumplen

entonces { P } **do** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **od** { Q } se cumple, suponiendo que la instrucción iterativa termina.

La definición formal de invariante viene dada por (ii), es decir, cualquier predicado P que satisfaga (ii) es un invariante de la instrucción iterativa.

Mostremos entonces formalmente que "**suma** = $(\sum j: 0 \leq j < i : j^2) \wedge 0 \leq i \leq N$" es un invariante para el programa en (1):

Probemos que $\{ P \wedge i < N \}$ **suma** := **suma** + i*i; i := i+1 { P } se cumple:

P(i := i+1)
≡ por sustitución y aritmética
suma = $(\sum j: 0 \leq j < i+1 : j^2) \wedge 0 \leq i+1 \leq N$

Tomamos este último predicado como postcondición de **suma** := **suma** + i*i:

(**suma** = $(\sum j: 0 \leq j < i+1 : j^2) \wedge 0 \leq i+1 \leq N$) (**suma** := **suma** + i*i)
≡ por sustitución
R**:** **suma** + i*i = $(\sum j: 0 \leq j < i+1 : j^2) \wedge 0 \leq i+1 \leq N$

¿ $[P \wedge i < N \Rightarrow R]$? o lo que es lo mismo:

¿ **suma** = $(\sum j: 0 \leq j < i : j^2) \wedge 0 \leq i \leq N \wedge i < N \Rightarrow$

suma $+ i*i = (\sum j: 0 \le j < i+1 : j^2) \wedge 0 \le i+1 \le N$ es una tautología ?

Demostración:

Supongamos que se cumple: **suma** $= (\sum j: 0 \le j < i : j^2) \wedge 0 \le i \le N \wedge i < N$. Entonces:

$0 \le i \le N \wedge i < N$
$\Rightarrow$ por aritmética
$0 \le i+1 \le N$

Por otro lado:

$(\sum j: 0 \le j < i+1 : j^2)$

= por separación del término de la suma j=i, y porque $0 \le i < i+1$ (existe al menos un término que separar, ya que $0 \le i$ es invariante)

$(\sum j: 0 \le j < i : j^2) + i^2$

= por hipótesis

suma $+ i^2$

La terminación de una instrucción iterativa se puede probar determinando una función entera sobre el espacio de estados que esté acotada inferiormente y que decrezca estrictamente en cada iteración, o esté acotada superiormente y que crezca estrictamente en cada iteración. Tal función la llamamos *función de cota*. Para el programa en (1) esta función es f(lista de variables) = i, pues de acuerdo al invariante P, i está acotado superiormente por N. Por otra parte, i crece estrictamente en cada iteración pues se le suma 1 en cada iteración (i := i+1), es decir, para cualquier constante C se tiene que:

$\{ P \wedge i < N \wedge i = C \}$ **suma** := **suma** + i*i; i := i+1 $\{ i > C \}$ se cumple

Note que hemos podido también tomar como función de cota a f(lista de variables)= N-i, la cual es estrictamente decreciente en cada iteración y acotada inferiormente por 0.

Finalmente, combinando la regla de la instrucción iterativa con el requerimiento de terminación, obtenemos la siguiente regla que permite demostrar la correctitud de una instrucción iterativa con dos guardias:

<u>Teorema de la Invariancia</u>:

Si
(i) $[P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q]$
(ii) $\{ P \wedge B_0\} S_0 \{ P \} \wedge \{ P \wedge B_1\} S_1 \{ P \}$ se cumplen
(iii) Existe una función entera t sobre el espacio de estados tal que:

a) $[P \wedge (B_0 \vee B_1) \Rightarrow t \geq 0]$,
b) $\{ P \wedge B_0 \wedge t = C \}\ S_0\ \{ t < C \}$, y
c) $\{ P \wedge B_1 \wedge t = C \}\ S_1\ \{ t < C \}$

entonces { P } **do** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **od** { Q } se cumple.

<u>Demostración del Teorema de la Invariancia</u>:

Sabemos de antes que suponiendo terminación, (i) y (ii) implican que se cumple:

{ P } **do** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **od** { Q }

Ahora demostraremos que (i), (ii) y (iii) implican terminación.

Haremos una demostración por el absurdo.

Supongamos que se cumple (i), (ii) y (iii) y sin embargo la instrucción iterativa no termina, es decir, el número de iteraciones es infinito.

Sea t_i el valor de t cuando concluye la iteración i. Como el número de iteraciones es infinito entonces la sucesión de números t_0, t_1, t_2, ... es infinita. Por (iii.b) y (iii.c) sabemos que esta sucesión es estrictamente decreciente. Por lo tanto existirá un k para el cual t_k es negativo. Por otro lado, que no termine la instrucción iterativa es equivalente a decir que al comienzo de cualquier iteración siempre será verdad $(P \wedge B_0) \vee (P \wedge B_1)$. Como P siempre es verdad al comienzo de cualquier iteración (por (i) y (ii)) entonces $(B_0 \vee B_1)$ siempre será verdad al comienzo de cualquier iteración.

Por otro lado, (iii.a) es equivalente a (pues $(p \Rightarrow q) \equiv (\neg q \Rightarrow \neg p)$ es una tautología):

$$t < 0 \Rightarrow \neg P \vee (\neg B_0 \wedge \neg B_1)$$

Como t_k es negativo entonces $\neg P \vee (\neg B_0 \wedge \neg B_1)$ es verdad. Como P es siempre verdad por (i) y (ii) entonces concluimos que $(\neg B_0 \wedge \neg B_1)$ es verdad. Esto último contradice lo que habíamos afirmado antes: que $(B_0 \vee B_1)$ siempre será verdad al comienzo de cualquier iteración.

En conclusión, si (i), (ii) y(iii) son verdad entonces la instrucción iterativa debe terminar, pues de lo contrario llegamos a una contradicción.

Las anotaciones para una instrucción iterativa son las siguientes:

{ Invariante: P, Cota: t }
do $B_0 \rightarrow \{ P \wedge B_0 \}\ S_0$ {P, Demostración 1}
[] $B_1 \rightarrow \{ P \wedge B_1 \}\ S_1$ {P, Demostración 2}
od

{ Q, Demostración 3, terminación: Demostración 4 }

donde:
Demostración 1: demostrar que $\{ P \wedge B_0 \} S_0 \{ P \}$ se cumple.
Demostración 2: demostrar que $\{ P \wedge B_1 \} S_1 \{ P \}$ se cumple.
Demostración 3: demostrar que $[P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q]$.
Demostración 4: demostrar que $[P \wedge (B_0 \vee B_1) \Rightarrow t \geq 0]$,
$\{ P \wedge B_0 \wedge t = C \} S_0 \{ t < C \}$ se cumple, y
$\{ P \wedge B_1 \wedge t = C \} S_1 \{ t < C \}$ se cumple

Frecuentemente, el invariante es la postcondición de una instrucción, que llamamos *instrucción de inicialización de P*, que precede la instrucción iterativa. Si S corresponde a tal instrucción y H es su precondición, las anotaciones serían:

{ H }
S
{ Invariante: P, Demostración 0, Cota: t }
do $B_0 \rightarrow \{ P \wedge B_0 \} S_0$ {P, Demostración 1}
[] $B_1 \rightarrow \{ P \wedge B_1 \} S_1$ {P, Demostración 2}
od
{ Q, Demostración 3, terminación: Demostración 4 }

donde Demostración 0 es una demostración de que { H} S { P } se cumple.

Al igual que en la instrucción de selección, queda sobreentendido que las expresiones B_0 y B_1 están bien definidas en los estados que satisfacen P. Por lo que otra demostración, de ameritarse, debe ser demostrar [P $\Rightarrow$ B_0 y B_1 están bien definidas], y será incluida en la demostración 3.

Ejercicios:

1) Haga las demostraciones demo0, demo1, demo2, demo3, demo4, en el programa siguiente, para demostrar que es correcto:

```
[ var x, y, N: int;
  { N ≥ 0 }
  x := 0;
  y := 0;
  { Invariante P: 0 ≤ x ∧ y ≤ N, demo 0, cota: x + 2(N-y) }
  do x ≠ 0 → { P ∧ x ≠ 0 } x := x-1 { P, demo1 }
  [] y ≠ N → { P ∧ y ≠ N } x := x + 1; y := y + 1 { P, demo2 }
  od
  { x = 0 ∧ y = N, demo3, terminación: demo4}
]
```

2) Analice los procesos iterativos descritos en los ejercicios 0,1,2,3,4, página 37

del Kaldewaij, determine un invariante, una función de cota y demuestre formalmente la correctitud de dichos programas.