7. Diseño de Algoritmos

Como ya hemos mencionado, el principio básico del diseño descendente es “tratar de resolver un problema mediante la resolución de problemas más simples”. En este capítulo ahondaremos en el diseño de algoritmos iterativos.

En la sección 7.1 presentamos estrategias de solución de problemas basadas en el diseño descendente. Presentaremos en la sección 7.2 esquemas de algoritmos básicos de tratamiento secuencial.

Como vimos en la sección 3.2, el diseño descendente también se denomina *técnica de refinamiento sucesivo*. Hasta el momento hemos aplicado el diseño descendente sólo para refinar acciones abstractas en términos de acciones cada vez menos abstractas, es decir, que se acercan cada vez más a las instrucciones propias de nuestro lenguaje de programación. Sin embargo, cuando hacemos un primer algoritmo para resolver un problema, las acciones describen realmente la interacción de los objetos involucrados en el enunciado del problema. Por lo tanto no basta con refinar las acciones entre objetos sino que es preciso refinar (o representar) los objetos abstractos en términos de objetos más concretos. Con frecuencia, los objetos involucrados en el enunciado del problema no son directamente representables en lenguajes de programación convencionales (por ejemplo, los conjuntos); de allí la necesidad de refinar datos. La representación de objetos abstractos en términos de objetos concretos lo llamamos *refinamiento de datos*, y presentaremos una metodología a seguir para llevar a cabo tal representación.

7.1. Diseño Descendente

La metodología de diseño descendente de programas consiste en:

1) Definir una solución de un problema en términos de la composición de soluciones de problemas que a priori son más sencillos de resolver, o de esquemas de solución ya conocidos.
2) La primera solución del problema corresponderá a una composición de acciones sobre los objetos al más alto nivel de abstracción, es decir, a los involucrados en la especificación del problema.
3) Aplicar refinamiento sucesivo, el cual consiste en refinar tanto las acciones como los datos hasta conseguir que el algoritmo inicial se convierta en un programa.

En esta sección ahondaremos en la práctica del diseño descendente, tanto en refinamiento de acciones como de datos. Hasta el momento hemos aplicado el diseño descendente sólo para refinar acciones abstractas en términos de acciones cada vez menos abstractas, es decir, que se acercan cada vez más a las instrucciones propias de nuestro lenguaje de programación. Sin embargo, cuando hacemos un primer algoritmo para resolver un problema, las acciones describen realmente la interacción de los objetos involucrados en el enunciado del problema. Por lo tanto no basta con refinar las acciones entre objetos sino que es preciso refinar (o representar) los objetos abstractos en términos de objetos más concretos, a esto último es a lo que llamamos *refinamiento de datos*. En los ejemplos de

diseño descendente vistos hasta ahora, no fue necesario refinar los datos pues estos correspondían a tipos no estructurados, como los números enteros, los booleanos, etc., considerados tipos básicos de nuestro pseudolenguaje que no hace falta refinar.

El principio básico del diseño descendente es "se debe programar hacia un lenguaje de programación y no en el lenguaje de programación", lo que significa partir de algoritmos en términos de los objetos involucrados en la especificación original del problema e ir refinándolos hasta obtener un programa en el lenguaje de programación que hayamos escogido. Una vez que un primer algoritmo correcto es encontrado, podemos cambiar la representación de los objetos por otros para mejorar, por ejemplo, la eficiencia (en el Capítulo 9 hablaremos de eficiencia) y/o implementarlos en el lenguaje de programación.

Cada objeto involucrado en la especificación de un problema posee una estructura y propiedades que lo caracteriza. Esta estructura y propiedades definen la *clase o el tipo del objeto*. Mas concretamente, una clase de objetos se define por un conjunto de valores y un comportamiento que viene expresado por operaciones sobre ese conjunto de valores; por ejemplo, el tipo número entero tiene como conjunto de valores {..., -1, 0, 1, 2, 3, ...} y como operaciones la suma, la resta, la multiplicación, etc. El tipo secuencia de caracteres tiene como conjunto de valores a las funciones de [0..n) en los caracteres, cualquiera sea el número natural n, y algunas operaciones son obtener el primero de la secuencia, insertar un elemento en una posición dada de una secuencia, etc.

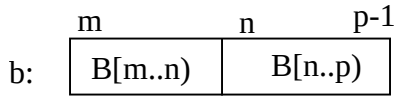
En un lenguaje de programación moderno, como JAVA, que permite *definir clases de objetos*, el refinamiento de datos se efectúa construyendo una unidad de programa aparte, llamada igualmente *clase*, que contiene segmentos de programa que implementan por una parte la representación de los objetos en términos de las estructuras de datos que ofrece el lenguaje y por otra parte, las operaciones de la clase. Decimos que JAVA permite *encapsular* la implementación de tipos abstractos de datos. El encapsulamiento de datos conlleva al *ocultamiento de datos*, es decir, el programador que sólo desea manipular objetos de un determinado tipo de datos no tiene por qué preocuparse por cómo se representa ese tipo en función de tipos concretos; lo que realmente le interesa es poder operar con objetos de ese tipo, y por lo tanto la implementación puede permanecer oculta al programador, permitiéndole así no involucrarse con los detalles de implementación del tipo.

7.1.1. Tratar de resolver un problema en términos de problemas más simples

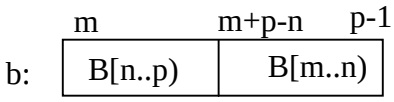
El término "más simple" puede significar diferentes cosas según el contexto. Un problema puede ser más simple debido a que algunas de sus restricciones han sido omitidas (resultando en una generalización del problema). El problema puede ser más simple porque, al contrario, le hemos agregado restricciones. Cualquiera sea el caso, la estrategia de resolver un problema en términos de problemas más simples consiste entonces primero que nada en tratar de identificar y resolver casos mas simples del mismo problema y tratar de resolver el problema original utilizando la solución de los casos más simples

Ejemplo:

Problema: Se quiere encontrar un programa que intercambie dos segmentos de un arreglo b con dominio [m..p), es decir, dados m< n < p y el arreglo b de la figura siguiente:



Donde B es el valor original del arreglo b. El arreglo b quedará modificado como en la figura siguiente:



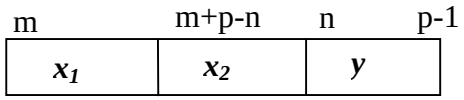
El programa sólo podrá declarar un número constante de variables adicionales de tipos básicos y utilizar sólo operaciones de intercambio de dos elementos de un arreglo más las propias a de los tipos básicos.

Solución:

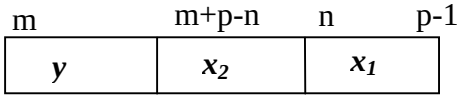
¿Cómo comenzaríamos?. Si los dos segmentos del arreglo son del mismo largo tendríamos un problema más simple que resolver (ejercicio: Hacer un procedimiento llamado *IntercambiarIguales* que intercambie dos segmentos disjuntos de igual largo de un arreglo, se pasará como parámetros el arreglo, el índice inicial de cada segmento y el largo de los segmentos a intercambiar). Supongamos que el procedimiento *IntercambiarIguales* consiste en intercambiar dos segmentos disjuntos de igual largo. Veamos si podemos resolver nuestro problema en términos de este problema más simple.

Suponga por el momento que el segmento b[m..n) es mas largo que b[n..p). Consideremos que el segmento b[m..n) consiste de dos segmentos, el primero de los cuales es del mismo largo que b[n..p) (ver diagrama (a) dado más abajo). Entonces los segmentos de igual largo x_1 y y pueden ser intercambiados obteniéndose el diagrama (b); además, el problema original puede ser resuelto intercambiando los segmentos x_2 y x_1. Estas dos secciones pueden ser de distintos largos, pero el largo del mayor segmento es menor que el largo del mayor segmento del problema original, por lo que hemos hecho algún progreso.

(a)



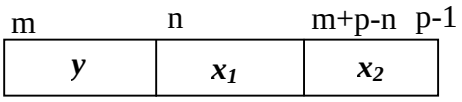
(b)



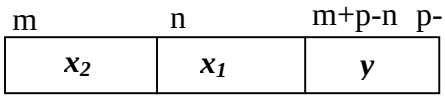
Ahora supongamos que en lugar del primer segmento del problema original, es el segundo segmento el que tiene largo mayor, es decir, b[n..p) es más grande. Este caso es ilustrado en

el diagrama (c), y el procedimiento *IntercambiarIguales* puede ser utilizado para transformar el arreglo como en el diagrama (d).

(c) b:



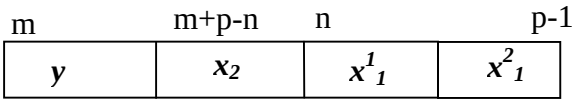
(d) b:



Ahora tratemos de llevar esta idea a un programa. Los diagramas (b) y (d) indican que después de ejecutar *IntercambiarIguales*, n siempre es el índice inferior de la sección más a la derecha a ser intercambiada. Lo cual es cierto también al comienzo.

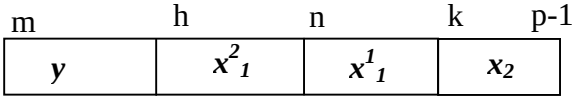
Supongamos que en (b) x_1 es mas largo que x_2, tendremos la situación siguiente:

(b') b:



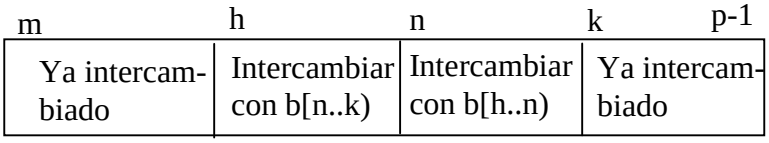
Donde $x_1 = x^1_1 || x^2_1$, x_2 y x^2_1 tienen el mismo largo. Después de intercambiar x_2 y x^2_1 mediante *IntercambiarIguales* obtenemos:

(b'') b:



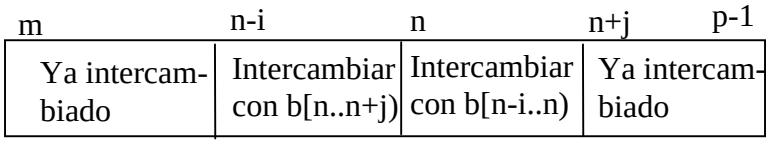
Note que los segmentos [m..h) y [k..p) ya están en el sitio que les corresponde, y hay que intercambiar los subsegmentos [h..n) y [n..k) del segmento [h..k). Por lo tanto, podemos obtener el invariante:

b:



Sin embargo, note que el algoritmo requiere la comparación de los largos de b[n..k) y b[h..n). También, el procedimiento *IntercambiarIguales* requiere los largos de los segmentos. Por lo tanto puede resultar mejor tomar los largos de los segmentos en lugar de sus índices extremos. Por lo que el invariante P se convierte en el predicado $0 < i \leq n\text{-}m \wedge 0< j \leq p\text{-}n$, junto con:

b:



Ejercicio: expresar el invariante P como un predicado.

Usando la función de cota decreciente t = max (i,j), el programa es como sigue:

```
[  const m, n, p: entero;
   var b: arreglo [m..p) de entero;
   var i, j: entero;
   { Precondición: m < n < p }
   i,j := n-m, p-n;
   { Invariante: P, cota: t }
   do  i > j → IntercambiarIguales(b, n-i, n, j); i := i-j
    []  i < j → IntercambiarIguales(b, n-i, n+j-i, i); j := j-i
   od;
   { P ∧ i = j }
   IntercambiarIguales(b, n-i, n, i)
   { Postcondición }
]
```

Como nota interesante, resulta que si eliminamos del programa anterior las llamadas al procedimiento *IntercambiarIguales*, el programa que resulta es el **algoritmo de Euclides para determinar el máximo común divisor, MCD(n-m, p-n), de n-m y p-n**; es decir, el máximo común divisor de los largos de los segmentos originales:

```
[  const m, n, p: entero;
   var i, j: entero;
   { Precondición: m < n < p }
   i,j := n-m, p-n;
   { Invariante: 0 < i  ∧  0 < j  ∧  MCD(n-m, p-n) = MCD(i,j) , cota: max(i,j) }
   do  i > j → i := i-j
       i < j → j := j-i
   od;
   { i = j = MCD(n-m,p-n) }
]
```

Ejercicios: página 225 del Gries.

7.1.2. Refinamiento de Datos

Veamos a través de un ejemplo sencillo en qué consiste el refinamiento de datos:

Problema: Hacer un procedimiento que calcule todas las soluciones complejas de la ecuación $Ax^2+Bx+C=0$, con A, B y C reales. 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:

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

Note que en la ecuación $A.x^2+B.x+C=0$, la operación suma (+) se realiza sobre números complejos, es decir, es la suma sobre números complejos, al igual que el producto (.). Por lo tanto nuestro problema involucra objetos que pertenecen a los tipos de datos (o clases): números reales y números complejos.

Nuestro pseudolenguaje nos permite hacer programas que manipulen tipos de datos cualesquiera, en particular el tipo de dato "número complejo". Sin embargo, cuando programamos en un lenguaje de programación particular, el refinamiento de datos es necesario llevarlo a cabo si nuestro lenguaje de programación no provee el tipo "número complejo". A continuación llevamos a cabo el desarrollo del programa en pseudolenguaje y utilizaremos refinamiento de datos para representar los números complejos mediante un par de números reales correspondientes a la parte real y la parte imaginaria del número complejo.

La especificación del programa sería:

[const A, B, C: real;
  var conj: Conjunto de Número complejo;
  { Pre: verdad }
  S
  {Post: conj = $\{x : (x \in$ Número complejo$) \wedge (A.x^2 + B.x + C = 0)\}$ }
]

Note que la especificación anterior involucra los tipos de datos "número real,", "número complejo" y "conjunto de números complejos".

Como no se impone ninguna condición adicional a los valores de los coeficientes del polinomio, sólo que sean números reales, debemos hacer un análisis por casos, pues dependiendo de los valores de los coeficientes tenemos diferentes maneras de proceder para calcular las raíces del polinomio. Por lo tanto, dividimos el espacio de estados en función de los coeficientes del polinomio y dependiendo del valor que éstos puedan tener, habrá una solución algorítmica 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:

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

Habrá dos raíces complejas si el discriminante, $B^2 - 4.A.C$, no es cero; en cuyo caso las

raíces complejas son: $\frac{-B+\sqrt{B^2-4.A.C}}{2.A}$ y $\frac{-B-\sqrt{B^2-4.A.C}}{2.A}$. Tendrá una sola raíz compleja si el discriminante es cero. Cuando A = 0 y B ≠ 0, el polinomio es de primer grado, y existirá una sola raíz. Y cuando A = 0, B = 0 y C = 0, todo número complejo es solución. Cuando A=0, B=0 y C ≠ 0, no existirá solución.

Por lo tanto podemos hacer una especificación más concreta donde introducimos una variable entera n que nos indica el número de soluciones que tiene la ecuación $A.x^2 + B.x + C=0$. Que no haya solución será equivalente a n=0. Hay una sola raíz es equivalente a n=1 y la variable x1 contendrá la raíz. Hay dos soluciones es equivalente a n=2 y las soluciones quedarán almacenadas en las variables x1 y x2. Todo número complejo es solución de la ecuación $A.x^2 + B.x + C=0$, si y sólo si n=3. Una especificación más concreta (o refinada) sería:

[const A, B, C: real;
  var x1, x2: Número complejo;
  var n: entero;
  { Pre: verdad }
  S
  {Post: (n=0 ∧ (∀x: x ∈ Número complejo: $A.x^2 + B.x + C \neq 0$))
        ∨
         (n=1 ∧ (∀x: x ∈ Número complejo: $A.x^2 + B.x + C = 0 \equiv x = x1$))
        ∨
         (n=2 ∧ (∀x: x ∈ Número complejo: $A.x^2 + B.x + C = 0 \equiv x = x1 \vee x = x2$))
               ∨
         (n=3 ∧ (∀x: x ∈ Número complejo: $A.x^2 + B.x + C = 0$)) }
]

Note que la variable “conj” desaparece en la nueva especificación, pues la teoría del problema nos permitió concluir que el número de soluciones es 0, 1, 2 ó infinito, y esto lo representamos mediante la variable entera n, y dos variables complejas x1 y x2. Por lo tanto, al conjunto “conj” de la especificación original lo hemos podido representar mediante las variables n, x1 y x2. Con esta nueva representación hemos hecho un refinamiento de datos. La relación existente entre “conj” y su representación en términos de n, x1 y x2, se denomina ***Invariante de Acoplamiento***, y este es:

(|conj|=0 ∧ n=0) ∨ (|conj|=1 ∧ n=1 ∧ conj={x1}) ∨
(|conj|=2 ∧ n=2 ∧ conj = {x1, x2}) ∨ (conj=conjunto de los números complejos ∧ n=3)

A efectos de desarrollar el programa, la postcondición Q0 ∨ Q1 ∨ Q2 ∨ Q3 puede ser reescrita en términos de A, B y C, con la finalidad de obtener una postcondición que nos permita desarrollar un programa.

La nueva especificación sería:
[const A, B, C: real;

var x1, x2: Número complejo;
var n: entero;
{ Pre: verdad }
S
{Post: (n = 0 $\wedge$ (A=0 $\wedge$ B=0 $\wedge$ C≠0)) $\vee$
(n = 1 $\wedge$ ((A = 0 $\wedge$ B ≠ 0) $\vee$ (A ≠ 0 $\wedge$ B^2- 4.A.C = 0))
$\wedge$ ($A.x1^2$ + B.x1 + C = 0)) $\vee$
(n = 2 $\wedge$ A ≠ 0 $\wedge$ ($A.x1^2$ + B.x1 + C = 0) $\wedge$ (B^2 – 4AC ≠ 0)
$\wedge$ ($A.x2^2$ + B.x2 + C = 0) $\wedge$ x1 ≠ x2) $\vee$
(n = 3 $\wedge$ A = 0 $\wedge$ B = 0 $\wedge$ C = 0) }
]

El programa completo sería:

```
[  const A, B, C: real;
   var x1, x2: Número complejo;
   var n: entero;
    var disc: real;

   { Pre: verdad }
    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 + sqrt(disc)) / (2*A);
                                       x2 := (-B - sqrt(disc)) / (2*A)
                         fi

    fi

   {Post: ( n = 0 ∧  (A=0 ∧ B=0 ∧ 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 ∧ ( A.x1² + B.x1 + C = 0 ) ∧ (B² – 4AC ≠ 0)
                                       ∧ ( A.x2² + B.x2 + C = 0 ) ∧ x1 ≠ x2 ) ∨
         ( n = 3 ∧ A = 0 ∧ B = 0 ∧ C = 0 ) }
]
```

Hemos introducido la variable "disc" con el propósito de aumentar la eficiencia del programa calculando una sola vez el discriminante de la ecuación de segundo grado.

En este punto hemos encontrado un algoritmo que resuelve el problema. El algoritmo manipula diferentes tipos de datos, entre los que se encuentra el tipo "número complejo". Si nuestro lenguaje de programación no posee el tipo "número complejo" debemos aplicar ***Refinamiento de Datos*** para representar el tipo número complejo en términos de los *tipos concretos de datos* de nuestro lenguaje de programación, es decir, los tipos que ofrece el lenguaje. Note que en la fase de especificación del problema también podemos hacer refinamiento de datos, tal y como sucedió en nuestro ejemplo, en donde representamos al conjunto solución "conj" por las variables n, x1 y x2.

El proceso de refinamiento de datos que se sigue para obtener un programa a partir del algoritmo anterior es el siguiente:

1) Representar el tipo abstracto de dato Número complejo en términos de tipos más concretos de datos. Esto significa representar los valores del tipo abstracto en términos de un conjunto de valores de tipos concretos. Por ejemplo, a un número complejo **x** podemos hacer corresponder un par de números reales **xr** y **xi** que representan respectivamente la parte real y la parte imaginaria del número complejo **x**. El ***Invariante de Acoplamiento*** sería: x = xr + xi*i. Note que hemos podido decidir representar a un número complejo mediante su representación polar (módulo y ángulo) en lugar de su representación cartesiana.

2) Utilizando el invariante de acoplamiento debemos implementar, mediante funciones y/o procedimientos, las operaciones asociadas al tipo de dato en términos de su representación por tipos concretos. Las operaciones a implementar deberán ser las operaciones del tipo que son utilizadas en nuestro algoritmo, aunque el conjunto de operaciones a implementar podría ser mayor si queremos que la implementación del tipo de dato sea utilizada por otros programas que utilicen esas operaciones sobre el tipo. Implementar una operación del tipo significa definir una función o un procedimiento por cada operación. Por ejemplo, si se suman dos números complejos en nuestro algoritmo, debemos hacer una función que reciba como parámetros de entrada dos números complejos representados por tipos concretos y devuelva el número complejo resultante de la suma, representado en términos de los tipos concretos.

Un número complejo *x* lo representaremos por un arreglo de dos números reales, llamémoslo *ax*, tal que se cumple el invariante de acoplamiento:

Invariante de acoplamiento: $x = ax[0] + ax[1] * i$

La única operación sobre números complejos que necesitamos en nuestro programa es poder crear un número complejo conociendo su parte real y su parte imaginaria. Para esto, definimos la función *ConstruirComplejo* como sigue:

arreglo de real fun ConstruirComplejo(**entrada** xr, xi: **real**)
{Pre: verdad}

{Post: Devuelve un arreglo, denotémoslo x, que representa al número complejo
x[0]+x[1]*i}

```
[
  var x: arreglo [0..2) de real;
  x[0] := xr;
  x[1] := x1;
  Devolver (x)
]
```

El programa completo sería:

```
[  const A, B, C: real;
   var x1a, x2a: arreglo [0..2) de real;
   var n: entero;
   var disc, raizdisc: real;

   { Pre: verdad }
    if
         (A = 0  ∧ B = 0 ∧ C = 0) → n := 3
    []   (A = 0  ∧ B = 0 ∧ C ≠ 0) → n := 0
    []   (A = 0  ∧ B ≠ 0 ) →   n := 1;
                               x1a := ConstruirComplejo(-C/B,0)
    []   (A ≠ 0) →      disc := B*B-4*A*C;
                        if  disc = 0 → n := 1;
                                       x1a := ConstruirComplejo(-B/(2*A),0)
                        []  disc > 0 → n := 2;
                                       raizdisc := RaizCuadrada(disc);
                                       x1a := ConstruirComplejo((-B + raizdisc)/(2*A),0);
                                       x2a := ConstruirComplejo((-B -  raizdisc)/(2*A),0)
                        []  disc < 0 → n := 2;
                                       raizdisc := RaizCuadrada(-disc);
                                       x1a := ConstruirComplejo(-B/(2*A),raizdisc/(2*A));
                                       x2a := ConstruirComplejo(-B/(2*A),-raizdisc/(2*A))
                        fi

    fi

   {Post: ( n = 0  ∧   (A=0 ∧ B=0 ∧ C≠0)  )   ∨
          ( n = 1  ∧  ( ( A = 0  ∧  B ≠ 0 )  ∨  (  A ≠ 0  ∧  B^2- 4.A.C = 0 ) )
                                  ∧ (x1 = x1a[0]+x1a[1].i)  ∧   ( A.x1^2 + B.x1 + C = 0 ) )      ∨
          ( n = 2  ∧  A ≠ 0  ∧  ( A.x1^2 + B.x1 + C = 0 ) ∧ (B^2 – 4AC ≠ 0)
                                  ∧ (x1 = x1a[0] + x1a[1].i) ∧ (x2 = x2a[0] + x2a[1].i)
                                  ∧   ( A.x2^2 + B.x2 + C = 0 ) ∧   x1 ≠ x2  )    ∨
          ( n = 3  ∧  A = 0 ∧ B = 0 ∧ C = 0 )   }
]
```

Note, por ejemplo, que como el trozo de programa original siguiente:

if disc = 0 → n := 1; x1 := -B/(2*A)
[] disc ≠ 0 → n := 2;

$$x1 := \frac{-B + \sqrt{disc}}{2 * A};$$

$$x2 := \frac{-B - \sqrt{disc}}{2 * A}$$

fi

satisface la especificación:

{ Pre: A ≠ 0 ∧ disc = B*B-4*A*C }
{ Post: n = 2 ∧ A ≠ 0 ∧ ($A.x1^2$ + B.x1 + C = 0) ∧ ($A.x2^2$ + B.x2 + C = 0) ∧ x1 ≠ x2}

entonces podemos demostrar, utilizando el invariante de acoplamiento, que el trozo de programa siguiente:

if disc = 0 → n := 1;
x1a := ConstruirComplejo**(**-B/(2*A),0)
[] disc > 0 → n := 2;
raizdisc := RaizCuadrada(disc);
x1a := ConstruirComplejo**(**(-B + raizdisc)/(2*A),0);
x2a := ConstruirComplejo**(**(-B - raizdisc)/(2*A),0)
[] disc < 0 → n := 2;
raizdisc := RaizCuadrada(-disc);
x1a := ConstruirComplejo**(**-B/(2*A),raizdisc/(2*A));
x2a := ConstruirComplejo**(**-B/(2*A),-raizdisc/(2*A))
fi

satisface la especificación:

{ Pre: A ≠ 0 ∧ disc = B*B-4*A*C }
{ Post: n = 2 ∧ A ≠ 0 ∧ ($A.x1^2$ + B.x1 + C = 0) ∧ ($A.x2^2$ + B.x2 + C = 0) ∧ x1 ≠ x2
∧ (x1 = x1a[0] + x1a[1].i) ∧ (x2 = x2a[0] + x2a[1].i)}

En el programa propuesto para hallar las soluciones complejas de la ecuación $A.x^2$ + B.x + C = 0, la representación de un número complejo en términos de tipos concretos no queda oculta al programador de la aplicación. Sin embargo, los constructores de procedimientos y funciones permiten implementar separadamente las operaciones del tipo; y un programador que las utilice le basta sólo con conocer la especificación de los procedimientos y funciones correspondientes. Lo ideal hubiese sido que el programador pudiese utilizar el tipo *Número complejo* sin tener que conocer como se representa en términos de tipos concretos (en nuestro caso un número complejo se representa por arreglo[0..2) de real). Una de las

ventajas de no tener que conocer la representación del tipo estriba en el hecho de poder modificar la representación del tipo *Número complejo* sin que esto afecte a los programas que usan el tipo *Número complejo*. Esto lo trataremos de subsanar en la sección 7.1.3. donde introduciremos un constructor de tipos de datos.

7.1.3. Encapsulamiento y Ocultamiento de Datos

Cuando **utilizamos** un tipo de dato, como "número complejo", no nos debería interesar cómo se representa el tipo en función de los tipos concretos. Lo que realmente nos debe interesar es poder operar con objetos de ese tipo. El programa se vuelve más complicado de entender cuando modificamos el algoritmo inicial para escribirlo en función de los tipos concretos de datos como hicimos en el ejemplo anterior.

Si el lenguaje de programación permite definir aparte los tipos abstractos de datos (*encapsulamiento de datos*) y que su implementación esté oculta al programador (*ocultamiento de datos*), entonces tendremos un mecanismo para realizar programas más claros. En JAVA, por ejemplo, existe el constructor CLASS (clase) que permite implementar encapsulamiento y ocultamiento de datos.

Introducimos a continuación un constructor de nuestro pseudolenguaje que nos permitirá implementar un tipo abstracto de datos. Este será un fragmento de programa que se define aparte del programa que lo usa.

Con este constructor lograremos:

- Definir y encapsular un tipo abstracto de datos.
- Ocultar la representación de un tipo abstracto de datos en términos de tipos concretos.
- Que los programas que hagamos sean más claros, al no tener que implementar los tipos abstractos de datos en el mismo fragmento de programa que los usa.
- La posibilidad de definir variables cuyo tipo sea el que definimos.
- Introducir la noción de *clases y objetos*. Nociones propias de la *programación orientada a objetos* (paradigma utilizado por el lenguaje de programación JAVA).

Procedamos a definir nuestro nuevo constructor, utilizando como ejemplo el tipo abstracto de dato *número complejo*. Un número complejo, es un tipo cuyos valores se pueden representar por dos números reales, la parte real y la parte imaginaria. Por otro lado, existen operaciones asociadas a números complejos como son sumar dos números complejos, multiplicarlos, determinar su parte real, determinar su parte imaginaria. Es decir, al tipo está asociado un conjunto de valores y un conjunto de operaciones aplicables a objetos de ese tipo. En programación orientada a objetos, una *clase* es el tipo de dato de un objeto, la cual incluye un conjunto de *atributos* que corresponden al valor de un objeto de la clase. Por otro lado, una clase posee *métodos*. Los métodos son procedimientos o funciones que están asociados a objetos de la clase. Por ejemplo, determinar la parte real de un número complejo puede corresponder a un método de la clase de los números complejos.

El constructor de tipos de nuestro pseudolenguaje permitirá declarar variables de la forma:

var *c1, c2, c3*: *NumeroComplejo*;

donde *NumeroComplejo* es el nombre de la clase que implementa a los números complejos.

Procederemos a hacer la especificación de la clase *NumeroComplejo* con las operaciones: construir un número complejo a partir de su parte real y su parte imaginaria, construir un número complejo a partir de su representación polar, obtener la parte real de un número complejo, obtener la parte imaginaria de un número complejo, sumar un número complejo a otro número complejo. Luego haremos una implementación de esta clase donde representamos a un número complejo por el par formado por su parte real y su parte imaginaria. Los nuevos términos que aparecen en la especificación serán explicados a lo largo de la sección.

La ***especificación*** de la clase *NumeroComplejo* es la siguiente:

clase *NumeroComplejo*

var *x:* número complejo;

constructor *NumeroComplejoCartesiano*(entrada *pr, pi*: real)
{Pre: verdad}
{Post: x = pr + pi*i }

constructor *NumeroComplejoPolar*(entrada *modulo, angulo*: real)
{Pre: *modulo* ≥ 0 ∧ 0 ≤ *angulo* < 360 }
{Post: x = (*modulo** *coseno*(*angulo*)) + (*modulo* * *seno*(*angulo*)) * *i* }

real metodo *ParteReal*( )
{Pre: verdad }
{Post: devuelve la parte real de *x* }

real metodo *ParteImaginaria*( )
{Pre: verdad }
{Post: devuelve la parte imaginaria de *x* }

metodo *Sumar*(entrada *otro: NumeroComplejo*)
{Pre: *x=A* }
{Post: *x = A + otro.x* }

finclase

La especificación anterior indica que la clase (o tipo abstracto de dato) *NumeroComplejo* estará formada por objetos cuyos valores son números complejos y cuyas operaciones son: *NumeroComplejoCartesiano, NumeroComplejoPolar, ParteReal, ParteImaginaria, Sumar*. Las operaciones *NumeroComplejoCartesiano* y *NumeroComplejoPolar* son operaciones

constructoras de la clase, es decir, permiten crear objetos tipo número complejo, de allí la palabra reservada "**constructor**" en la especificación. Las otras operaciones se definen con la palabra reservada "**metodo**" para indicar que son operaciones que se aplican a objetos de la clase.

Por lo tanto, con la declaración:

var *c*: *NumeroComplejo*;

estamos indicando que *c* es un objeto de tipo *NumeroComplejo*. Y podemos asignar un valor a *c* con la siguiente instrucción:

c := *NumeroComplejoCartesiano*(2,10)

Después de que se ejecute esta instrucción, *c* tendrá el valor 2 + 10*i. Este será el valor del atributo *x* de *c*. Si denotamos el atributo *x* de *c* por *c.x*, tendremos *c.x* = 2+10*i después de ejecutarse la instrucción anterior.

Implementación y refinamiento de la clase *NumeroComplejo* en el pseudolenguaje:

Ahora procederemos a realizar un refinamiento de datos en la definición de la clase *NumeroComplejo*, mediante la representación de un número complejo por el par formado por su parte real y su parte imaginaria.

Los atributos de la nueva clase incluyen los atributos asociados a la representación de los valores del tipo en términos de tipos concretos y cuya relación se establece mediante el invariante de acoplamiento. En nuestro caso los atributos de un objeto de la clase *NumeroComplejo* serán su parte real y su parte imaginaria. Denotemos estos atributos por *preal* y *pimag* respectivamente. Parte de la clase se escribe:

clase *NumeroComplejo*
  var *preal, pimag: real*; .....

Por lo tanto el **invariante de acoplamiento** entre la especificación inicial y la implementación que estamos desarrollando sería:

*x = preal + pimag * i*

donde *x* es el atributo de la especificación y, *preal* y *pimag* son los atributos de la implementación.

Así la clase refinada e implementada es la siguiente:

clase *NumeroComplejo*

  var *preal, pimag:* **real**;

{ **Invariante de acoplamiento**: *x = preal + pimag * i* }

constructor *NumeroComplejoCartesiano*(entrada *pr, pi*: real)
{Pre: verdad}
{Post: *preal = pr* $\wedge$ *pimag = pi* }
[
preal := pr; pimag := pi
]

constructor *NumeroComplejoPolar*(entrada *modulo, angulo*: real)
{Pre: *modulo* $\geq 0 \wedge 0 \leq$ *angulo* < 360 }
{Post: *preal = (modulo** cos*(angulo)) $\wedge$ pimag = (modulo** sen*(angulo))* }
[
preal = (modulo coseno(angulo)); pimag = (modulo * seno(angulo))*
]

real metodo *ParteReal*( )
{Pre: verdad }
{Post: devuelve *preal* }
[Devolver (*preal*)]

real metodo *ParteImaginaria*( )
{Pre: verdad }
{Post: devuelve *pimag* }
[Devolver (*pimag*)]

metodo *Sumar*(entrada *otro: NumeroComplejo*)
{Pre: *preal= A* $\wedge$ *pimag = B* }
{Post: *preal = A + otro.preal* $\wedge$ *pimag = B + otro.pimag* }
[
preal := preal + otro.preal;
pimag := pimag + otro.pimag
]

finclase

Las normas de estilo de programación orientada a objetos obligan a que un atributo de representación de un objeto no sea visible a los programas que usan dicho objeto. Esto es para reforzar la técnica de ocultamiento de datos: no importa cómo se implementa un tipo abstracto de datos, lo que importa es su comportamiento para poder usarlo, y por lo tanto, no se debe tener acceso a la representación interna de sus atributos. Por lo que no podemos hacer referencia a la parte real de un objeto *c1* tipo *NumeroComplejo,* en un programa que usa dicho objeto. La expresión *c1.preal* no puede aparecer en otro trozo de programa que no sea la implementación de la clase *NumeroComplejo*.

Note, sin embargo, que dentro de la implementación de la clase *NumeroComplejo*, en el método *Sumar*, hemos accedido directamente a los atributos del objeto *otro*; esto se debe a que *otro* es tipo *NumeroComplejo* y dentro de la implementación de una clase se permite tener acceso a los atributos de otros objetos de la misma clase.

El ocultamiento de los atributos de representación permite también que sea posible modificar más adelante la implementación de *NumeroComplejo* (por ejemplo, representándolo en coordenadas polares) sin que esto afecte a los programas que usen el tipo *NumeroComplejo*.

Si queremos obtener la parte real del número complejo *c1*, la instrucción de llamada a método (análoga a llamada a procedimiento o función) sería *c1.ParteReal*( ), la cual corresponde en este caso a una llamada a una función asociada a objetos de la clase *NumeroComplejo*, y que devuelve la parte real del objeto *c1*. Si queremos la parte imaginaria del objeto *c1*, escribiríamos *c1.ParteImaginaria( )*.

Note que las llamadas a métodos arriba indicadas han sido aplicadas al objeto *c1*. En programación NO orientada a objetos, este objeto *c1* sería el primer argumento o parámetro de tales procedimientos o funciones. En programación orientada a objetos, se dice que *c1* es el objeto receptor de la llamada. Por lo tanto, al apegarnos a la filosofía de orientación a objetos, el método *ParteReal* tendrá un parámetro formal implícito adicional que corresponderá al número complejo al que se le aplica la llamada. Siguiendo la convención utilizada por muchos lenguajes orientados a objetos (entre ellos JAVA), a tal parámetro implícito lo llamaremos *this*. En este momento podemos observar la diferencia existente entre un procedimiento o función convencional y un método de una clase: una llamada a método (por ejemplo, *c1.ParteReal( )*) debe ir asociada a un objeto; decimos que la llamada a método es un *mensaje* que estamos pasando al objeto receptor. Por lo tanto, en la sintaxis de la instrucción de llamada a método debe aparecer explícitamente el objeto receptor del mensaje (en el ejemplo anterior es *c1*).

El objeto receptor *this* de una llamada a método siempre es un parámetro formal de **entrada-salida**. Por lo tanto, los métodos tienen un argumento implícito adicional que de haberse indicado explícitamente, tendría la siguiente declaración:

entrada-salida *this: NumeroComplejo*

Por lo tanto, la instrucción:

x := c1.ParteReal( )

equivaldría, si hacemos explícito el parámetro asociado al objeto, a:

x := ParteReal(*c1*)

Sin embargo, esta última notación no es correcta para los métodos definidos en una clase.

Podemos hacer mención explícita de *this* en la implementación de la clase *NumeroComplejo*. Por ejemplo el método *ParteReal( )* lo hemos podido escribir:

real metodo *ParteReal*( )
{Pre: verdad }
{Post: devuelve *this.preal* }
[Devolver (*this.preal*)]

En general, el proceso de refinamiento, ocultamiento y encapsulamiento de datos consiste en especificar el tipo abstracto de dato y luego implementarlo en el lenguaje de programación utilizando las herramientas de encapsulamiento del lenguaje. Los programas que utilicen el tipo implementado lo harán mediante la declaración de variables que representarán objetos del tipo implementado y mediante paso de mensajes a objetos del tipo.

Veamos el programa completo refinado para el cálculo de las soluciones complejas de la ecuación $Ax^2+Bx+C = 0$, utilizando la clase *NumeroComplejo* (colocamos en itálicas los elementos asociados a la clase *NumeroComplejo*):

```
[  const A, B, C: real;
   var x1, x2: NumeroComplejo;
   var n: entero;
   var disc, raizdisc: real;

   { Pre: verdad }
    if
         (A = 0  ∧ B = 0 ∧ C = 0) → n := 3
   []    (A = 0  ∧ B = 0 ∧ C ≠ 0) → n := 0
   []    (A = 0  ∧ B ≠ 0 ) →   n := 1;
                               x1 := NumeroComplejoCartesiano(-C/B,0)
   []    (A ≠ 0) →       disc := B*B-4*A*C;
                         if  disc = 0 → n := 1;
                                        x1 := NumeroComplejoCartesiano(-B/(2*A),0)
                         [] disc ≠ 0 → n := 2;
                                        if  disc > 0  → raizdisc := RaizCuadrada(disc);
                                                        x1 := NumeroComplejoCartesiano(
                                                                (-B+raizdisc)/(2*A),0);
                                                        x2 := NumeroComplejoCartesiano(
                                                                (-B-raizdisc)/(2*A),0)
                                                [] disc < 0  → raizdisc := RaizCuadrada(-disc);
                                                        x1 :=
                                                        NumeroComplejoCartesiano(-B/(2*A),
                                                                raizdisc/(2*A));
                                                        x2 :=
                                                        NumeroComplejoCartesiano(-B/(2*A),
                                                                - raizdisc/(2*A));
```

fi

fi

fi

{Post: (n = 0 $\wedge$ (A=0 $\wedge$ B=0 $\wedge$ C$\neq$0)) $\vee$
(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)) $\vee$
(n = 2 $\wedge$ A $\neq$ 0 $\wedge$ ($A.x1^2$ + B.x1 + C = 0)
$\wedge$ ($A.x2^2$ + B.x2 + C = 0) $\wedge$ x1 $\neq$ x2) $\vee$
(n = 3 $\wedge$ A = 0 $\wedge$ B = 0 $\wedge$ C = 0) }
]

Otro ejemplo de Refinamiento de Datos: Registro estudiantil

Un registro estudiantil consiste de un conjunto de datos sobre estudiantes. Por cada estudiante, el registro estudiantil contiene el número de carnet del estudiante y su índice académico. Se quiere implementar la siguiente aplicación: dado un registro estudiantil y una lista de N estudiantes, determinar si entre los estudiantes de la lista se encuentra un estudiante con mayor índice del registro estudiantil. Se deberá crear la clase *RegistroEstudiantil* que permita definir objetos de este tipo. A efectos de la aplicación, el registro estudiantil estará creado y contiene al menos un estudiante, lista de estudiantes estará almacenada en un arreglo de enteros de largo N que contiene los carnets de los estudiantes.

La clase *RegistroEstudiantil* podría tener las siguientes operaciones: agregar un estudiante al registro estudiantil, dar el índice de un estudiante, determinar un estudiante con mejor índice, verificar si el registro tiene estudiantes, verificar si el registro tiene un estudiante dado; aparte de los constructores. Por otro lado, suponemos que el número de estudiantes que están en el registro está acotado por un cierto número MAX.

La especificación de esta clase sería:

clase *RegistroEstudiantil*
var *carnets*: conjunto de enteros;
var *indices*: función de enteros a reales;
const *MAX*: entero;

{**Invariante de Clase**: $(\forall c: c \in carnets: c > 0) \wedge (\forall i: i \in rango(indices): 1 \leq i \leq 5) \wedge$
$(dom(indices) = carnets) \wedge |carnets| \leq MAX$ }

constructor *RegistroEstudiantil*( )
{Pre: *verdad*}
{Post: $carnets = \varnothing \wedge indices = \varnothing$ }

metodo *AgregarEstudiante*(**entrada** *nuevoc*: **entero**; **entrada** *nuevoi*: **real**)

{Pre: *nuevoc* > *0* $\wedge$ *1* $\leq$ *nuevoi* $\leq$ *5* $\wedge$ *carnets = C* $\wedge$ *indices = I* $\wedge$ $|$*carnets*$|$ < *MAX* $\wedge$
nuevoc $\notin$ *carnets* }
{Post: *carnets = C* $\cup$ *{nuevoc}* $\wedge$ *indices = I* $\cup$ *{(nuevoc, nuevoi)}*}

real metodo *DarIndice*(**entrada** *c*: **entero**)
no modifica: *carnets, indices*
{Pre: *c* $\in$ *carnets* }
{Post: *devuelve indices (c)* }

metodo *UnMejorEstudiante*(**salida** *c*: **entero**; **salida** *i*: **real**)
no modifica: *carnets, indices*
{Pre: *carnets* $\neq \emptyset$}
{Post: *c* $\in$ *carnets* $\wedge$ *i = indices (c)* $\wedge$ ($\forall$ *otroc: otroc* $\in$ *carnets: indices(otroc)* $\leq$ *i)* }

booleano metodo *HayEstudiantes*( )
no modifica: *carnets, indices*
{Pre: verdad }
{Post: *Devuelve el valor de la expresión (carnets* $\neq \emptyset$) }

booleano metodo *EstaEstudiante*(entrada c: entero)
no modifica: *carnets, indices, MAX*
{Pre: verdad }
{Post: Devuelve *(c* $\in$ *carnets)* }

finclase

Note que la clase *RegistroEstudiantil* tiene más de un atributo, a diferencia de la clase *NumeroComplejo*. El **invariante de clase** premite establecer las relaciones que deben existir entre los atributos de la clase. Introducimos la frase reservada "**no modifica**" para indicar que el método no modificará los atributos que aparecen en la lista, y así abreviar el uso de variables de especificación. Una implementación de la clase *RegistroEstudiantil* es:

clase *RegistroEstudiantil*

var *seccarnets*: **arreglo [0..***MAX***) de entero**;
var *secindices*: **arreglo [0..***MAX***) de real**;
var *cantidad*: **entero**;
const *MAX*: **entero**;

{**Invariante de Clase**: 0 $\leq$ *cantidad* $\leq$ MAX $\wedge$
($\forall$i: 0 $\leq$ i < *cantidad*: *seccarnets*[i] > 0 $\wedge$ 1 $\leq$ *secindices*[i] $\leq$ 5) $\wedge$
($\forall$i: 0 $\leq$ i < *cantidad* $\wedge$ 0 $\leq$ j < *cantidad*:
i $\neq$ j $\Rightarrow$ *seccarnets*[i] $\neq$*seccarnets*[j]) }

{**Invariante de Acoplamiento**: carnets = { *seccarnets*[i] : 0 $\leq$ i < *cantidad* }$\wedge$

indices = { (*seccarnets*[i],*secindices*[I]) : 0 ≤ i < *cantidad* }

}

constructor *RegistroEstudiantil*( )
{Pre: *verdad* }
{Post: *cantidad = 0* }
[
cantidad := 0
]

metodo *AgregarEstudiante*(**entrada** *nuevoc*: **entero**; **entrada** *nuevoi*: **real**)
no modifica: *MAX*
{Pre: *nuevoc > 0 ∧ 1 ≤ nuevoi ≤5 ∧ seccarnets = SC ∧ sec indices = SI ∧ cantidad= C*
∧ cantidad < MAX ∧ (∀i: 0≤i<cantidad: secarnets[i] ≠ nuevoc) }
{Post: *seccarnets = SC(C:nuevoc) ∧ sec indices = SI(C:nuevoi) ∧ cantidad* = C + 1 }
[
seccarnets[cantidad] := nuevoc;
secindices[cantidad]:=nuevoi;
cantidad := cantidad + 1;
]

real metodo *DarIndice*(**entrada** *c*: **entero**)
no modifica: *seccarnets, secindices, MAX, cantidad*
{Pre: *(∃j: 0≤j <cantidad: seccarnets[j] = c)* }
{Post: Devuelve *z, tal que: (∃j: 0≤j <cantidad: seccarnets[j] = c ∧ secindices[j]=z)* }

[var *j:* entero;
j := 0;
do *(seccarnets[j] ≠ c) → j:= j+1*
Devolver (*secindices[j]*)
]

metodo *UnMejorEstudiante*(**salida** *c*: **entero**; **salida** *i*: **real**)
no modifica: *seccarnets, secindices, MAX,cantidad*
{Pre: *cantidad* ≠ 0}
{Post: *(∃j: 0≤j <cantidad: seccarnets[j] = c ∧ secindices[j]= i ∧*
(∀k: 0≤k < cantidad: secindices[k]≤ i)) }
[
var *k, j* : entero;
k := 1; *j* := 0;
do *k ≠ cantidad* → if *secindices[k] > secindices[j] → j:= k*
[] *secindices[k] ≤ secindices[j]* → skip
fi;
k := k+1
od;
c := *seccarnets*[j];

```
  i:= secindices[j]
 ]

 booleano metodo HayEstudiantes( )
 no modifica: seccarnets, secindices, MAX, cantidad
 {Pre: verdad }
 {Post:  Devuelve el valor de la expresión (cantidad ≠ 0) }
 [
  Devolver(cantidad ≠ 0)
 ]

 booleano metodo EstaEstudiante(entrada c: entero )
 no modifica: seccarnets, secindices, MAX, cantidad
 {Pre: verdad }
 {Post:  Devuelve (∃j: 0≤j <cantidad: seccarnets[j] = c)}
 [
  var j : entero;
  j := 0;  b:= verdad;
  do ( j ≠ cantidad) ∧ b → if  (seccarnets[j] ≠ c) → j:= j+1
                           [] (seccarnets[j] = c) → b:=falso
                           fi
  do;
  Devolver (j < cantidad)
 ]
```

finclase

La aplicación, utilizando la clase *RegistroEstudiantil* sería:

```
[
 const Reg: RegistroEstudiantil;
 const estudiantes: arreglo [0..N) de entero;
 const N: entero;
 var carnet_mejor, j : entero; var indice_mejor, indice_est: real;
 var esta: booleano;

 { Pre: N ≥ 0 ∧ (∀ j: 0 ≤ j < N : Reg.EstaEstudiante(estudiantes[j])) }

 esta := falso;
 Reg.UnMejorEstudiante(carnet_mejor, indice_mejor);
 j := 0;
 do (j ≠ N) ∧ (¬ esta) →  indice_est := Reg.DarIndice(estudiantes[j] ) ;
                          if  ( ind_est  = indice_mejor)  → esta := verdad
                          [] ( ind_est  ≠ indice_mejor)  → skip
                          fi;
                          j:=j+1
```

do
{Post: esta ≡ *(∃j: 0≤j <N: (∀z: z ∈ Z : Reg.EstaEstudiante(z) ⇒*
(Reg.DarIndice(z) ≤Reg.DarIndice(estudiantes[j]))) }
]

Ejercicios:

1) Aplicar encapsulamiento de datos los tipos Relación Binaria y Máquina de trazados.
2) Dado un conjunto finito A de números enteros se quiere hacer un programa que construya dos conjuntos, uno que contenga los números negativos de A y otros con los números no negativos de A. (ayuda: representar el tipo conjunto mediante arreglos).
3) Ejercicios pag. 117 Castro.

7.2. Esquemas de recorrido y búsqueda secuencial

Hasta ahora hemos considerado dos formas básicas de acceder a los datos, como son el acceso secuencial y el acceso directo. En el capítulo 6 hemos visto algunos algoritmos que acceden ya sea secuencial o directamente los elementos de un arreglo.

La diferencia entre acceso secuencial y directo se puede ilustrar con un ejemplo matemático. Recordemos que los números de Fibonacci se definen como la sucesión: 0, 1, 1, 2, 3, 5, 8,... donde el elemento n-ésimo, Fib(n), es la suma de Fib(n-1) y Fib(n-2) (para n mayor que 1). Si deseamos obtener el n-ésimo número de Fibonacci debemos partir de los elementos Fib(0) y Fib(1) e ir obteniendo cada vez el siguiente de la secuencia como suma de los dos anteriores, hasta llegar a construir el n-ésimo. Este método de obtener el n-ésimo número de Fibonacci, obteniendo los valores de la secuencia uno tras otro se denomina acceso secuencial: para acceder a un término se debe haber accedido antes a los términos anteriores. Ahora bien, si conociéramos una fórmula cerrada, función de n solamente, para determinar el n-ésimo número de Fibonacci, entonces no tenemos que hacer un acceso secuencial de la sucesión para obtenerlo, bastaría sólo con aplicar la fórmula para obtener el número. En este último caso, tenemos un acceso directo al n-ésimo número de Fibonacci, es decir, conocemos su valor a partir solamente de su posición, n, en la sucesión.

Otros ejemplos de acceso secuencial y directo son: búsqueda de una escena en una cinta de video (acceso secuencial), reproducción de una canción específica en un disco compacto (acceso directo). Utilizaremos el tipo archivo secuencial y el tipo arreglo para ilustrar los esquemas básicos de algoritmos para acceso secuencial.

7.2.1. Los Tipos Abstractos Archivo Secuencial de Entrada y Archivo Secuencial de salida

Un archivo secuencial de entrada es fundamentalmente una secuencia de objetos de un mismo tipo base, donde el acceso a cada objeto sólo puede llevarse a cabo de manera secuencial, es decir, para acceder al n-ésimo objeto, hay que recorrer los n-1 objetos anteriores a él en la secuencia. Por otro lado, el largo de la secuencia no es conocido a

priori. Un ejemplo de archivo secuencial de entrada es la entrada estándar de un computador como el teclado; también, un archivo almacenado en una cinta magnética, archivo estándar de texto.

El valor de un objeto del tipo Archivo Secuencial de Entrada está compuesto por:

1) Una secuencia S de elementos de tipo base "tipo base".
2) Un índice i, que llamaremos "el elemento índice del archivo secuencial", que representará el elemento del archivo al que se puede tener acceso en una instrucción de acceso al siguiente elemento de la secuencia.

Un archivo secuencial de entrada sólo tiene operaciones que permiten tener acceso secuencialmente a cada uno de los elementos de la secuencia comenzando desde el primero hasta el último. Como no conoceremos de antemano el largo del archivo (es decir, de la secuencia asociada al archivo), tendremos un operador (FinArchivo()) que nos indica si se ha alcanzado el fin del archivo (es decir, de la secuencia).

A.S denotará la secuencia, A.i denotará el elemento índice del archivo secuencial A de entrada.

Las operaciones sobre archivos secuenciales son:

- AbrirEntrada(A,s): Es un constructor y permite crear un archivo con secuencia s. El valor de la secuencia A.S del archivo será la secuencia s de tipo "tipo base" y el elemento índice A.i tendrá valor 0.

- FinArchivo(A): es una función booleana. Devuelve "verdad" si y sólo si se ha alcanzado el fin del archivo, es decir, si el índice A.i de A es igual al largo de la secuencia A.S.

- Leer(A,x): coloca en x el elemento de A.S cuyo índice es A.i e incrementa en 1 a A.i. Esta operación es aplicable si y sólo si A.i < $\left|A.S\right|$

Note que el elemento índice A.i de un archivo secuencial A, divide implícitamente a la secuencia A.S en dos partes: el segmento A.S[0..i) que llamaremos parte izquierda de A y denotaremos por pi(A) y el segmento A.S[i..N), donde N es el largo de S, que llamaremos parte derecha de A y denotaremos por pd(A). Note que basta con conocer exactamente uno de los tres valores: pi(A), A.i ó pd(A), para que los otros dos queden precisamente determinados. Note además que si en un programa utilizamos un archivo de entrada A, en cualquier momento de la ejecución del programa pi(A) representa los elementos que han sido "leídos" por el programa, es decir, los elementos que han sido obtenidos mediante una operación Leer(...), y este hecho lo podemos utilizar en las aserciones de las aplicaciones que usen archivos secuenciales de entrada.

La especificación de la clase Archivo de Entrada es:

clase ArchivoEntrada

var S: secuencia de tipo base;
var i: entero;

{ Invariante de clase: $0 \leq i \leq |S|$ }

constructor AbrirEntrada(entrada s:secuencia de objetos de tipo base)
{ Pre: verdad}
{ $S = s \wedge i = 0$ }

booleano metodo FinArchivo( )
no modifica: i, S
{ Pre: verdad}
{ Devuelve ($i = |S|$) }

metodo Leer(salida x: tipo base)
no modifica: S
{Pre: $i < |S| \wedge i = I$ }
{Post: $x = S[I] \wedge i = I+1$ }

finclase

Un archivo secuencial de salida sólo tiene operaciones que permiten construir su secuencia asociada, es decir, permiten "escribir" elementos al final de la secuencia. No se permite leer elementos en archivos de salida. Ejemplo de archivo de salida es la salida estándar del computador, por ejemplo, una ventana de interfaz de comandos.

Las operaciones sobre archivos de salida son:

- AbrirSalida(A,s): Es un constructor y permite crear un archivo con secuencia s. El valor de la secuencia A.S del archivo será la secuencia s de tipo "tipo base".

- Escribir(A,x): agrega como último elemento de la secuencia A.S, un nuevo elemento cuyo valor será el del objeto x.

La especificación de la clase ArchivoSalida es:

clase ArchivoSalida

var S: secuencia de tipo base;

constructor AbrirSalida(entrada s:secuencia de objetos de tipo base)
{ Pre: verdad}
{ $S = s$ }

metodo Escribir(entrada x: tipo base)
{Pre: S= X }
{Post: S = X||<x> }

finclase

7.2.2. Acceso Secuencial: Esquemas de recorrido y búsqueda secuencial

Veamos dos problemas sencillos que permiten esquematizar los modelos fundamentales de algoritmos que actúan sobre secuencias:

1) Sumar todos los valores de una secuencia de enteros.
2) Verificar si aparece la letra 'w' en una frase.

En el primer problema debemos recorrer todos los elementos de la secuencia y a cada elemento hacerle un mismo tratamiento. Otros problemas que tienen estas mismas características son: cálculo del máximo número de una secuencia de enteros, contar el número de caracteres iguales a 'f' en una secuencia de caracteres. Diremos que estos problemas los podemos resolver con un **esquema de recorrido secuencial**, que ya hemos visto en el caso de arreglos y que daremos mas adelante para el tipo archivo secuencial.

En el segundo problema la estrategia correcta de solución debería ser que al momento en que se encuentra el elemento buscado no debe continuarse el recorrido de la secuencia, de forma que sólo se llega al final de la secuencia si el elemento buscado no aparece en ésta. Diremos que este problema lo podemos resolver con un **esquema de búsqueda secuencial**, que daremos mas adelante.

Ahora bien, pueden existir situaciones en donde no conocemos el largo de las secuencias que queremos tratar. Por lo tanto necesitamos alguna forma de identificar cuándo hemos tratado el último elemento de la secuencia. Si la secuencia viene representada por un archivo secuencial de entrada, el operador FinArchivo( ) nos indica cuando hemos alcanzado el final de la secuencia.

Esquemas de recorrido secuencial:

Supongamos que tenemos la secuencia S de largo N, y queremos sumar los elementos de S, el problema lo podemos resolver como sigue:

[const N: entero; { $N \geq 0$ }
const B: secuencia de entero;
var suma: entero;
{ $|S| = N$ }
suma := 0;
k := 0;
{Invariante: (suma = ($\sum i : 0 \leq i < k: S[i])$)$\wedge(0 \leq k \leq N)$;
Cota decreciente: N-k }

```
  do k ≠ N
        → suma := suma + S[k];
           k := k+1
  od
   { suma = ( ∑i : 0 ≤ i < N: S[i])   }
]
```

Podemos abstraer los elementos básicos de un algoritmo de recorrido: dar un mismo tratamiento a cada elemento de la secuencia, recorriendo la secuencia del primero al último.

Por lo tanto, si la especificación de un problema es como sigue:

{ S es una secuencia de elementos de largo N}
recorrido
{ Se trataron de la misma forma todos los elementos de S}

Entonces la solución obedecerá al siguiente esquema de programa, conocido como **esquema de recorrido secuencial**:

Declaración de variables
Inicializar tratamiento (entre otras cosas, dar acceso al primer elemento de la secuencia, por ejemplo, inicializar i en 0)
{ Inv: (han sido tratados los elementos de S[0..i)
  Cota: $|S| - i$}
do (no se ha alcanzado el final de S, por ejemplo, i ≠ largo de S)→
   Tratar elemento S[i];
   Acceder al siguiente elemento (es decir incrementar i en 1)
od
Tratamiento final

<u>Aplicación de esquemas de recorrido secuencial cuando representamos el tipo secuencia mediante el tipo archivo secuencial de entrada:</u>

Supongamos que en un archivo secuencial de entrada tenemos una secuencia de números enteros y queremos calcular la suma de los elementos del archivo secuencial. La solución sería utilizar el esquema de recorrido secuencial:

```
[ const S: secuencia de entero;
  var A: ArchivoEntrada;
  var suma, x: entero;
  { verdad }
  A.AbrirEntrada(S);
  suma := 0;
  {Invariante: (suma = ( ∑j : 0 ≤ j < |pi(A)| : pi(A)[j]))  ∧ S = pi(A)||pd(A);
   Cota decreciente: |S| - |pi(A)|  }
  do ¬ FinArchivo(A)  → A.Leer(x);
```

```
                                suma := suma + x;
  od
   { suma = ( ∑i : 0 ≤ i < |S| : S[i])   }
]
```

En el invariante del algoritmo anterior $|pi(A)|$ es igual a A.i; sin embargo, no utilizamos A.i por el principio de ocultamiento de datos, es decir, no conocemos la representación interna de un archivo secuencial de entrada.

Esquemas de búsqueda secuencial:

Resolvamos el problema de determinar si un texto S contiene la letra 'w', lo cual nos ayudará a abstraer el esquema correspondiente. La especificación sería:

```
[ const S: secuencia de caracter;
  var esta: booleano;
  { verdad }
   buscar
  { esta ≡ (∃i: 0 ≤ i < |S| : S[i] = 'w' )  }
]
```

Una solución sería:

```
[  const N: entero;
  const S: secuencia de caracter;
  var esta: booleano;
  { N ≥ 0 ∧ (|S| = N) }
  esta :=  falso;
  k := 0;
  { Invariante: ( esta ≡ (∃i: 0 ≤ i < k: S[i] = 'w' ) ) ∧ (0≤k≤N); Cota: N-k }
  do k ≠ N →  esta := esta ∨ (S[k] = 'w' );
                k := k+1
  od
  { esta ≡ (∃i: 0 ≤ i < |S| : S[i] = 'w' )  }
]
```

Para no tener que recorrer toda la secuencia, habiendo ya encontrado el elemento buscado, el esquema asociado sería el siguiente:

```
[  const N: entero;
  const S: secuencia de caracter;
  var esta: booleano;
  { N ≥ 0 ∧ (|S| = N) }
  esta :=  falso;
  k := 0;
  { Invariante: ( esta ≡ (∃i: 0 ≤ i < k: S[i] = 'w' ) ) ∧ (0≤k≤N); Cota: N-k }
```

```
  do k ≠ N ∧ ¬esta
        → esta := (S[k] = 'w' );
            k := k+1
  od
  { esta ≡ (∃i: 0 ≤ i < |S|: S[i] = 'w' )  }
]
```

Si la secuencia no es vacía, tenemos la solución:

```
[  const N: entero;
  const S: secuencia de caracter;
  var esta: booleano;
  { N > 0  ∧ (|S| = N) }
  k := 0;
  { Invariante:  ¬(∃i: 0 ≤ i < k: S[i] = 'w' ) ∧ (0≤k≤N-1) ; Cota: N-k }
  do k ≠ N-1 ∧  S[k] ≠ 'w'
        →  k := k+1
  od
  esta := (S[k] = 'w')
  { esta = (∃i: 0 ≤ i < |S|: S[i] = 'w' )  }
]
```

Note que en la guardia del programa anterior no podemos colocar k ≠ N pues cuando k es N el valor S[N] no está definido y daría error. El programa anterior no funciona para secuencias de largo 0, mientras que el que lo precede sí.

Por lo tanto, si la especificación de un problema es como sigue:

{ S es una secuencia de elementos}
búsqueda
{ esta = existe un elemento x de S que cumple la propiedad P(x) }

Entonces la solución obedecerá al siguiente esquema de programa, conocido como **esquema de búsqueda secuencial**:

Declaración de variables;
Inicializar tratamiento (entre otras cosas, inicializar **i** en 0 y **esta** en falso);
{ Inv: ($\exists S_1,S_2$: (S = S_1 || S_2) ∧ (**esta** = existe un elemento en S_1 que cumple P(x)) ∧ (**i** = $|S_1|$));
  Cota: |S| - i }
do i ≠ |S| ∧ ¬ esta →
   esta := P(S[i]);
   Acceder al siguiente elemento (es decir incrementar i en 1)
od;
Tratamiento final

Aplicación de esquemas de búsqueda secuencial cuando representamos el tipo secuencia mediante el tipo archivo secuencial de entrada:

Problema: dado un archivo secuencial de entrada se quiere verificar si el archivo contiene la letra 'w'

```
[  const S: secuencia de caracter;
   var A: ArchivoEntrada;
   var esta: booleano;
  var x: carácter;
   { verdad }
   A.AbrirEntrada(S);
   esta :=  falso;
   {Invariante: ( esta ≡ (∃i: 0 ≤ i < |pi(A)|: pi(A)[i] = 'w' ) ) ∧ S = pi(A)||pd(A);
    Cota decreciente: |pi(A)| }
   do ¬ FinArchivo(A)  ∧ ¬ esta → A.Leer(x);
                                  esta := (x = 'w' );
   od
   { esta ≡ (∃i: 0 ≤ i < |S|: S[i] = 'w' )  }
]
```

Ejercicios:

1) Dado un archivo secuencial de caracteres hacer un programa que verifique si el archivo contiene el carácter 'w'. Considere dos casos: que exista o no centinela.
2) Hacer un programa que cree un archivo secuencial que contenga los cuadrados de los primeros n números naturales.
3) Dado un archivo secuencial de caracteres, formular y discutir varios programas que permitan encontrar el número de ocurrencias de la subsecuencia "hola" (Piense en una versión donde la idea abstracta es contar las ocurrencias del objeto "hola" en una secuencia de objetos, los objetos pueden ser palabras de largo 4, y así aplicar el esquema de recorrido secuencial visto en esta sección).
4) Hacer un programa que lea dos archivos secuenciales donde los elementos están ordenados en forma ascendente y cree un archivo secuencial con todos los elementos de los dos archivos originales ordenados en forma ascendente

Otros ejemplos de diseño descendente: Tratamiento de parejas de caracteres (pag. 92-100 Castro) para mostrar diseño descendente utilizando esquemas de recorrido y refinamiento de datos.