Introducción a la Programación

Oscar J. Meza H.[1]

Jesús Ravelo[1]

Departamento de Computación y Tecnología de la Información
Universidad Simón Bolívar
2012

[1] e-mails: meza@ldc.usb.ve, jravelo@ldc.usb.ve

Contenido

1. Introducción

Estas notas son una Introducción a la Resolución Sistemática de Problemas mediante Algoritmos. El objetivo de las notas es presentar una metodología que permita obtener soluciones algorítmicas correctas de un problema dado.

La Programación de computadoras comenzó como un arte, y aún hoy en día mucha gente aprende a programar sólo mirando a otros proceder (ej. Un profesor, un amigo, etc.), y mediante el hábito, sin conocimiento de los principios que hay detrás de esta actividad. En los últimos años, sin embargo, la investigación en el área ha arrojado teoría suficiente para que estemos en capacidad de comenzar a enseñar estos principios en los cursos básicos de enseñanza de la programación.

La metodología que proponemos toca la raíz de los problemas actuales en programación y provee principios básicos para superarlos. Uno de estos problemas es que los programadores han tenido poco conocimiento de lo que significa que un programa sea correcto y de cómo probar que un programa es correcto. Cuando decimos "probar" no queremos decir necesariamente que tengamos que usar un formalismo o las matemáticas, lo que queremos expresar es "encontrar un argumento que convenza al lector de la veracidad de algo". Desafortunadamente, los programadores no son muy adeptos a esto, basta observar la cantidad de tiempo que se gasta en "debugging" (proceso de depuración de programas para eliminar entre otros errores, los errores de lógica), y esto ¿por qué? Porque el método de programación normalmente usado ha sido "desarrollo por casos de prueba", es decir, el programa es desarrollado sobre la base de algunos ejemplos de lo que el programa debe hacer. A medida que se encuentran más casos de prueba, estos son ejecutados y el programa es modificado para que tome en cuenta los nuevos casos de prueba. El proceso continúa con modificaciones del programa a cada paso, hasta que se piensa que ya han sido considerados suficientes casos de prueba.

Gran parte del problema se ha debido a que no hemos poseído herramientas adecuadas. El razonamiento sobre cómo programar se ha basado en cómo los programas son ejecutados, y los argumentos sobre la corrección (diremos de ahora en adelante "correctitud", pues el término corrección se entiende como si quisiéramos corregir un programa y lo que queremos es probar que un programa es correcto) de un programa se han basado en sólo determinar casos de prueba que son probados corriendo el programa o simulados a mano. La intuición y el sentido común han sido simplemente insuficientes. Por otro lado, no siempre ha estado claro lo que significa que un programa sea "correcto", en parte porque la especificación de programas ha sido también un tema muy impreciso.

El principio básico de estas notas será el desarrollo simultáneo de un programa con la demostración de la correctitud. Es muy difícil probar la correctitud de un programa ya hecho, por lo que es más conveniente usar las ideas de prueba de correctitud a medida que se desarrolla el programa. Por otra parte, si sabemos que un programa es correcto, no tendrá sentido hacer muchas pruebas al programa con casos de prueba, sólo se harán las necesarias para corroborar que no cometimos un error, como humanos que somos, por ejemplo, en la transcripción del programa; además , así también reduciremos la cantidad de tiempo

dedicada a la tediosa tarea del "debugging". Para ello trataremos el tema de especificación de programas (o problemas) y el concepto de prueba formal de programas a partir del significado formal (la semántica) de los constructores básicos (es decir, las instrucciones del lenguaje de programación) que utilizaremos para hacer los programas.

Conscientes de que estas notas van dirigidas a un primer curso sobre enseñanza de la programación, el enfoque que llevaremos a cabo trata de balancear el uso de la intuición y el sentido común, con técnicas más formales de desarrollo de programas.

1.1. Problemas, algoritmos y programas

Un *problema* es una situación donde se desea algo sin poder ver inmediatamente la serie de *acciones* a efectuar para obtener ese algo.

Resolver un problema consiste primero que nada en comprender de qué trata y *especificarlo* lo más formalmente posible con el propósito de eliminar la *ambigüedad*, la *inconsistencia* y la *incompletitud*, con miras a obtener un enunciado claro, preciso, conciso y que capture todos los requerimientos. Normalmente la especificación en lenguaje natural lleva consigo problemas de inconsistencia, ambigüedad e incompletitud; es por esto que nos valemos de lenguajes más formales como las matemáticas (lógica, teoría de conjuntos, álgebra, etc.) para evitar estos problemas.

Ejemplo de ambigüedad: la frase "Mira al hombre en el patio con un telescopio" se presta a ambigüedad. ¿Utilizamos un telescopio para mirar al hombre en el patio o el hombre que miramos tiene un telescopio?

Ejemplo de inconsistencia: Por inconsistencia entendemos que en el enunciado podamos incurrir en contradicciones. Ejemplo sobre el procesador de palabras:
- Todas las líneas de texto tienen la misma longitud, indicada por el usuario.
- Un cambio de línea debe ocurrir solo después de una palabra, a menos que el usuario pida explícitamente la división por sílabas.

La inconsistencia resulta al preguntarse ¿Si la palabra es más larga que la línea? Entonces si el usuario no pide explícitamente la división por sílabas, esta línea tendrá mayor longitud.

Ejemplo de incompletitud: Por incompletitud entendemos que no todos los términos estén bien definidos o que no estén capturados todos los requerimientos del problema. En un manual de un procesador de palabras encontramos la frase "Seleccionar es el proceso de designar las áreas de su documento sobre las cuales desea trabajar" ¿qué significa designar, colocar el "ratón" dónde? ¿qué significa área? ¿cómo deben ser las áreas? ¿es una sola área?

En otras palabras, se trata de encontrar una representación del problema donde todo esté dicho, sea mediante gráficos, o utilizando otro lenguaje distinto al natural (ej. las matemáticas). En esta etapa interviene el proceso de abstracción, que permite simplificar el problema, buscando modelos abstractos equivalentes y eliminando informaciones superfluas.

Un problema no estará del todo bien comprendido si no hemos encontrado una representación en la cual todos los elementos que intervienen sean representados sin redundancia, sin ambigüedad y sin inconsistencias. El universo de búsqueda de la solución estará en ese momento bien delimitado y con frecuencia surgirá en forma más clara la dificultad principal del problema. El problema se convierte en más abstracto y más puro. Hablamos entonces de un *enunciado cerrado*.

Por ejemplo, si queremos determinar la ruta más corta para ir de Caracas a Barquisimeto en automóvil, podemos tomar el mapa vial de Venezuela y modelarlo a través de un grafo como en la figura 1, donde los nodos o vértices serían las ciudades y los arcos las vías de comunicación existentes entre las ciudades. A cada arco asociamos el número de kilómetros de la vía que representa y luego intentamos buscar la solución en el grafo.

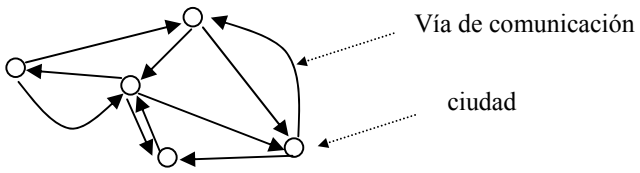


Figura 1

En matemáticas la forma más general de enunciado de un problema se puede escribir como: "Encontrar en un conjunto X dado, los elementos x que satisfacen un conjunto dado de restricciones K(x)". Por ejemplo: encontrar en el conjunto de los números naturales los números x tales que $x^2 + 83 = 37x$.

Observación:

- Cuando damos el espacio X, generalmente damos implícitamente la estructura de X y las operaciones lícitas sobre X. El conocimiento de X es un condensado crucial de información.

Una vez que obtenemos una especificación del problema que corresponda a un enunciado cerrado, en la práctica lo que hacemos es precisar aún más la especificación mediante la descomposición del problema en subproblemas más concretos (hacemos un *diseño descendente* del problema), hasta llegar a un punto del refinamiento del enunciado en donde sabemos llevar a cabo las *acciones* en él descritas. En ese momento tenemos la solución del problema, ya sea porque encontramos los resultados buscados o porque obtenemos *un algoritmo* que resuelve el problema, es decir, una descripción de cómo llevar a cabo un conjunto de acciones, que sabemos realizar a priori, sobre los datos del problema, para obtener los resultados buscados; en este último caso decimos que tenemos *una solución algorítmica del problema*.

Por ejemplo: "Determinar x en los números enteros tal que $x + 3 = 2$"

Este enunciado es equivalente a "Determinar x en los números enteros tal que x + 3 – 3 = 2 – 3" porque conocemos las propiedades del conjunto de los números enteros. Pero este enunciado nos lleva, por aplicación de las propiedades de números enteros (son estas propiedades las que consideramos cruciales conocer para resolver el problema), al siguiente enunciado que es la solución del problema: "Determinar x en los números enteros tal que x = -1 ". En este caso obtenemos directamente el número buscado.

Nuestro interés será buscar la solución de un problema como una descripción de una *acción* que manipula los datos hasta llegar al resultado, a esta descripción la llamamos *algoritmo*. *Una acción* es un evento producido por un actor, que llamamos el ejecutante; por ejemplo, una calculadora puede llevar a cabo la acción de sumar dos números, la calculadora sería el ejecutante de esa acción.

Una acción toma lugar durante un período de tiempo finito y produce un resultado *bien definido y previsto*. La acción requiere de la existencia de datos sobre los cuales se ejecuta para producir nuevos datos que reflejan el efecto esperado. Los datos que nos da el enunciado del problema poseen normalmente una estructura que los caracteriza (ej., son números, conjuntos, secuencias, etc.) y que llamaremos *el tipo del dato o la clase del dato*.

Una clase o tipo de dato viene dado por un conjunto de valores y su comportamiento, es decir, un conjunto de operaciones que podemos realizar con estos valores (por ejemplo, sobre los números naturales podemos sumar dos números, restar, multiplicar, etc.).

Un objeto lo podemos caracterizar por su *identificación*, su *clase ó tipo* y su *valor*. Decimos que un objeto es un *ejemplar (o instancia) de su clase*. Podemos decir que un algoritmo, en vez de manipular datos, manipulará objetos. Por ejemplo: en el problema de la búsqueda del camino más corto entre dos ciudades de Venezuela dado antes, el grafo es la clase de datos involucrada en la especificación final. Un grafo tiene operaciones bien definidas que podemos realizar sobre él, como por ejemplo: agregar un nodo, eliminar un arco, etc. El mapa de Venezuela y las distancias entre ciudades son los datos del problema y si vemos el mapa como un grafo, este mapa particular será un *objeto de la clase grafo*.

El conjunto de valores de los objetos manipulados por una acción, observados a un instante de tiempo dado t del desarrollo de ésta, es llamado el *estado* de los objetos manipulados por la acción en el instante t. Los valores de los objetos al comienzo de la acción los llamamos *los datos de entrada de la acción* y definen el estado inicial. Los valores de los objetos al concluir la acción los llamamos *datos de salida o resultados* y definen el estado final.

Si una acción puede ser descompuesta en un conjunto de acciones a realizar de manera secuencial, entonces diremos que la acción es un *proceso secuencial*. Un proceso secuencial no es más que una secuencia de acciones (o eventos). Retomando la definición del término algoritmo, podemos decir que *un algoritmo* es una descripción de un esquema o patrón de realización de un proceso o conjunto de procesos similares mediante un repertorio finito y no ambiguo de acciones elementales que se supone realizables a priori. Extenderemos el término acción a la descripción misma del evento y diremos que el algoritmo es la acción que hay que llevar a cabo para resolver el problema. Cuando un

algoritmo está expresado en términos de acciones de un lenguaje de programación, diremos que es un *programa*.

Por ejemplo: hacer una torta es una acción, esa acción se descompone en varias acciones más simples (agregar los ingredientes, mezclarlos, etc.) que serían el proceso que se lleva a cabo. La receta de la torta es el algoritmo o la descripción del proceso a seguir para hacer la torta.

Cuando decimos "conjunto de procesos similares" queremos decir que un algoritmo normalmente describe un proceso no sólo para un conjunto de datos de entrada específicos, sino para todos los posibles valores que puedan tener los objetos involucrados. Por ejemplo, cuando describimos el algoritmo de la multiplicación de dos números naturales no lo hacemos para dos números específicos (por ejemplo, 4 y 29) sino de una manera más general, de forma que describa la multiplicación de cualesquiera dos números naturales.

Ejemplo:

Enunciado del problema: Un vehículo que se desplaza a una velocidad constante v, consume l litros de gasolina cuando recorre k kilómetros. Determinar la cantidad r de gasolina que consumiría el vehículo a la velocidad v si recorre x kilómetros.

Note que el problema tiene un enunciado más general que el problema que consiste en buscar la ruta mínima entre dos ciudades específicas de Venezuela, en el sentido que independientemente de los valores de v, k, l y x (implícitamente cada una de las cantidades deben ser no negativas), queremos determinar un algoritmo que resuelva el problema (que calcule la cantidad r). Por lo tanto un mismo algoritmo deberá resolver el problema para valores particulares de v, k, l y x. Podemos ver a v, k, l, x, r como variables (al igual que en matemáticas) que pueden contener valores de un determinado tipo o clase de datos. En este caso el tipo de dato de todas estas variables es "número real".

De igual forma, podemos formular un problema más general que el dado sobre la ruta mínima entre dos ciudades de Venezuela, de la siguiente forma: Queremos buscar la ruta mínima para ir de una ciudad X a una ciudad Y en un país Z. En este caso tendremos tres variables que representan los objetos involucrados en el problema, a saber, dos ciudades y un país. Normalmente los problemas se presentan de esta forma. Es más útil encontrar un algoritmo para multiplicar dos números cualesquiera, que un algoritmo que sirva para multiplicar 2 por 3 solamente!

Ejemplos sobre correctitud versus análisis de casos de prueba

Primer ejemplo:

Ejemplo que muestra que la intuición no basta para desarrollar un algoritmo que resuelve un problema:

Se quiere pelar un número “suficiente” de papas que están en un cesto. El cesto de papas puede vaciarse en cualquier momento y podemos reponer el cesto con papas una vez éste se vacíe.

Solución:

Versión 1:

Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Si el cesto no está vacío entonces pelar una papa

Este algoritmo funciona cuando el número de papas a pelar es menor o igual al número de papas en el cesto. Si el cesto posee menos papas que las requeridas entonces caemos en un ciclo infinito (proceso sin fin) una vez que se vacíe el cesto

Versión 2:

Mientras el cesto no esté vació haga lo siguiente:
    Pelar una papa
    Pelar una papa

Se pelan tantas papas como hay en el cesto si el número de papas originalmente es par. Si el número inicial de papas es impar entonces hay una acción en el proceso que no puede ser ejecutada. Sólo cuando el cesto tiene el número de papas requerido el algoritmo resuelve el problema.

Versión 3:

Mientras el cesto no esté vacío y el número de papas peladas sea insuficiente haga lo siguiente: Pelar una papa

Este algoritmo no cae en ciclo infinito ni tratará de ejecutar una acción que es imposible de realizar. Sin embargo, solo cuando el cesto contiene el número de papas requerido el algoritmo resuelve el problema.

Versión 4:

Mientras el cesto no esté vacío hacer lo siguiente:
    Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
        Pelar una papa

Hagamos un diagrama de estados a medida que se ejecuta el proceso partiendo de todos los posibles estados iniciales. Este diagrama también se conoce como “corrida en frío” o “simulación del algoritmo”.

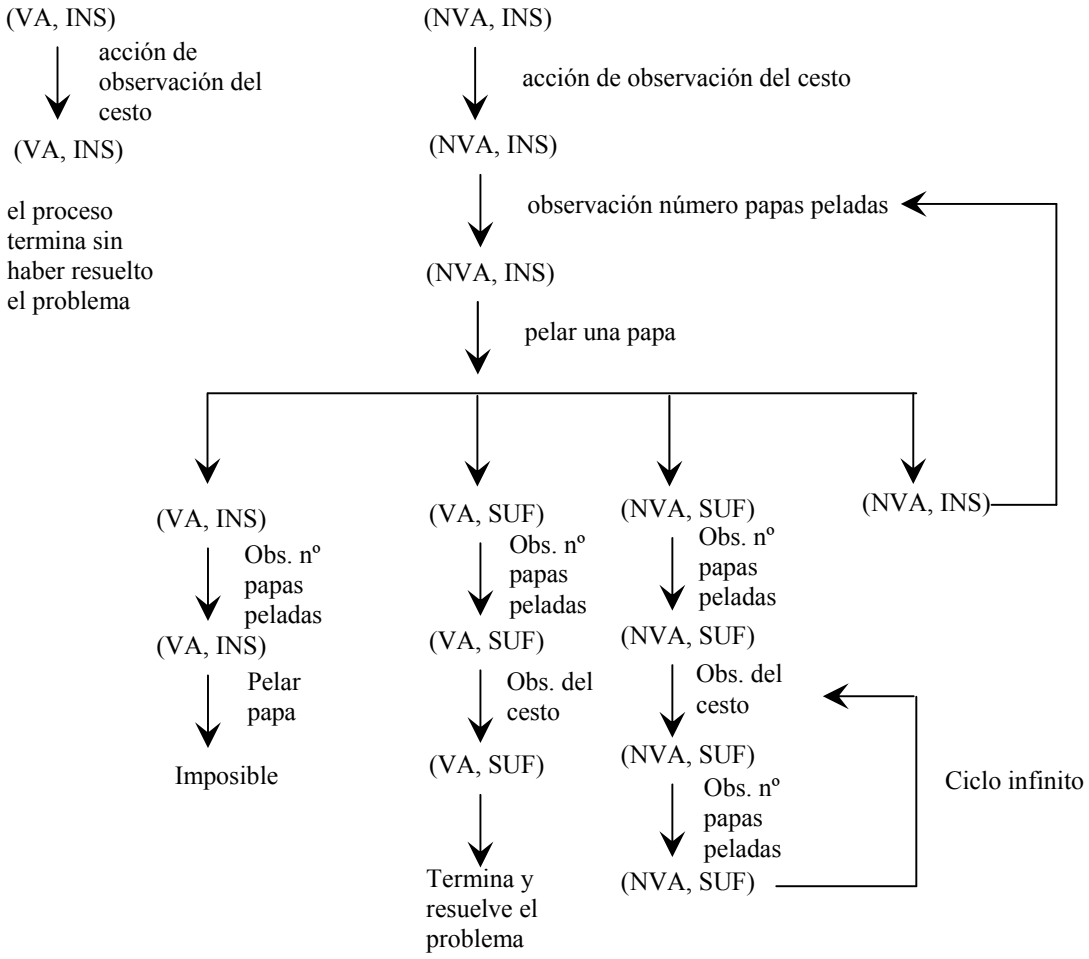


Figura 2

Descripción de los estados:

VA = cesto vacío,
INS = Número insuficiente de papas peladas

NVA = cesto no vacío
SUF = número suficiente de papas peladas

Estados iniciales posibles: (VA, INS) y (NVA, INS)

En las figura 2 vemos el proceso que genera el algoritmo partiendo respectivamente de los estados iniciales (VA, INS) y (NVA, INS)

Versión 5:

Mientras el número de papas no sea suficiente hacer lo siguiente:
Si el cesto no está vacío entonces pelar una papa
en caso contrario llenar el cesto con papas

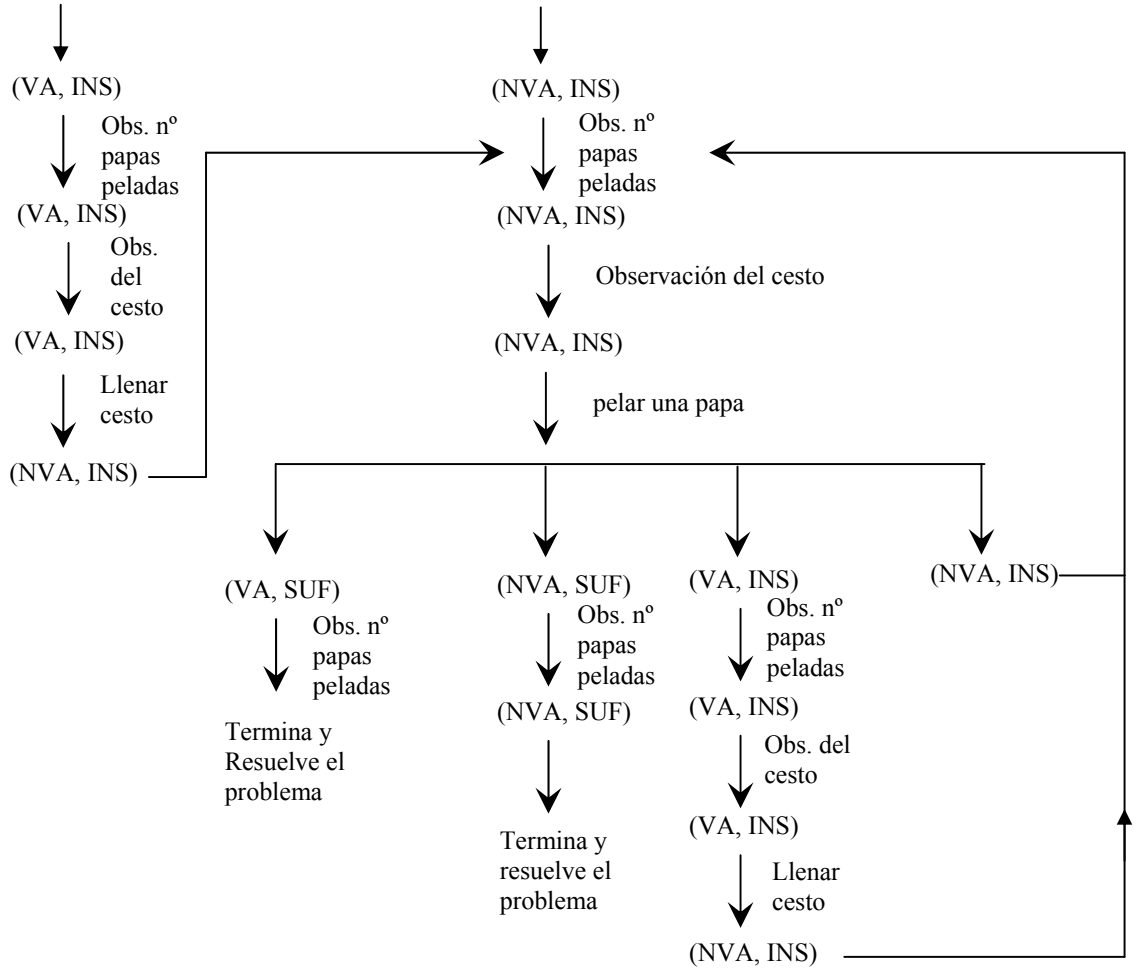


Figura 3

Esta versión es correcta, es decir, es un algoritmo que resuelve el problema, no existen acciones imposibles de realizar en la ejecución, ni cae en ciclo infinito y cuando termina se habrán pelado el número de papas requerido. El diagrama de estados de la figura 3 confirma este hecho.

Demostrar la correctitud del algoritmo anterior significa demostrar que partiendo de cualquier estado, al ejecutar el algoritmo, éste termina en un tiempo finito y cuando termina, este resuelve el problema (el estado final será SUF para el número de papas peladas).

Para este problema en particular, el diagrama de estados nos permite verificar la correctitud de nuestro problema y esto se debe a que el número posible de estados es muy reducido y podemos hacer una corrida completa para todos los estados iniciales posibles. Este no será el caso general, y no podremos utilizar el diagrama de estados (o corrida en frío) para verificar la correctitud del algoritmo pues el espacio de estados será, en general, infinito.

Parte de la verificación de la correctitud debe ser "el algoritmo termina", o sea, que el algoritmo no debe caer en ciclos infinitos. Vemos que el diagrama de estados, en este caso, nos permite comprobar esto, ya que todo ciclo pasa por la acción "pelar papa" y como el número de papas que se quiere pelar es finito, estos ciclos serán finitos. Y cuando el algoritmo termina, se vé que termina en un estado que resuelve el problema.

Versión 6:

En la vesión anterior hay un problema de *estilo*: la acción que se repite no siempre es "pelar papa", pero lo lógico es pelar una papa cada vez que se observa que hay un número insuficiente de papas.

Una versión "mejor":

Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Si el cesto está vacío entonces llenar el cesto con papas
    Pelar una papa

Las acciones que se repiten (el cuerpo del mientras) tienen un resultado previsible cada vez que se ejecutan: independientemente del estado inicial de la acción, al concluir ésta, se ha pelado una papa. En la versión (5) al concluir la acción que se repite no sabremos si se ha pelado una papa o no.

Ejercicio: Dibuje el diagrama de estados de la versión (6) y verifique que el algoritmo es correcto.

Estudie de la misma manera las siguientes versiones:

a) Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Mientras el cesto no esté vacío hacer:
        Pelar una papa

b) Mientras el número de papas peladas sea insuficiente o el cesto no esté vacío hacer:
    Pelar una papa

c) Si el cesto está vacío entonces llenarlo
    Pelar una papa
    Mientras el número de papas peladas sea insuficiente hacer:
        Si el cesto no está vacío entonces pelar una papa
        Si el cesto está vacío entonces llenar el cesto y pelar una papa

    ¿En esta versión las acciones que se repiten tienen siempre como estado final haber pelado exactamente una papa?. ¿Cree usted que esta versión es poco elegante? ¿Por qué?.

Segundo ejemplo (importancia de las especificaciones y del razonamiento sobre programas):

Supongamos que hemos escrito un programa muy largo que, entre otras cosas, calcula, como resultados intermedios, el cociente **q** y el resto **r** de dividir un entero no negativo **x** entre un entero positivo **y**. Por ejemplo, para **x**=7 e **y**=2, el programa calcula **q**=3 y **r**=1.

El trozo de programa que calcula el cociente y el resto es muy simple, pues consiste en el proceso de sustraer tantas veces **y** a una copia de **x,** guardando el número de sustracciones que se hayan efectuado, hasta que una sustracción más resulte en un número negativo. El trozo de programa, tal y como aparece en el programa completo es el siguiente (los puntos "..." son para indicar que el trozo de programa está inserto en otro programa más grande):

```
...
r := x;  q:=0;
mientras r>y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
...
```

Comencemos a depurar el programa.

Respecto al cálculo del cociente y el resto, sabemos claramente que inicialmente el divisor no debe ser negativo y que después de concluir la ejecución del trozo de programa, las variables deberían satisfacer:

$$x = y*q + r$$

De acuerdo a lo anterior, agregamos algunas instrucciones para imprimir algunas variables con la finalidad de revisar los cálculos:

```
 ...
escribir ('dividendo x =', x, 'divisor y =', y);
r := x;  q:=0;
mientras r>y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
escribir ( ' y*q + r = ', y*q + r);
...
```

Desafortunadamente, como este trozo de programa se ejecuta muchas veces en una corrida del programa, la salida es muy voluminosa. En realidad queremos conocer los valores de las variables sólo si ocurre un error. Supongamos que nuestro lenguaje de programación permite evaluar expresiones lógicas cuando estas aparecen entre llaves, y si la evaluación de la expresión lógica resulta falsa entonces el programa se detiene e imprime el estado de las variables en ese momento; si la evaluación de la expresión es verdadera, entonces el programa continúa su ejecución normalmente. Estas expresiones lógicas son llamadas *aserciones,* debido a que estamos afirmando que ellas serán verdaderas en el instante justo después de la ejecución de la instrucción anterior a ella.

A pesar de la ineficiencia que significa insertar aserciones, pues aumenta el número de cálculos que debe hacer el programa, decidimos incorporarlas pues es preferible que el programa detecte un error cuando sea utilizado por otros a que arroje resultados erróneos.

Por lo tanto incorporamos las aserciones al programa, y eliminamos las instrucciones de escritura:

```
  ...
{ y > 0 }
r := x;  q:=0;
mientras r > y hacer
        comienzo  r := r-y;  q := q+1  fin;
{  x = y*q + r }
...
```

De esta forma, nos ahorramos largos listados (papel impreso) en el proceso de depuración.

En una corrida de prueba, el chequeo de aserciones detecta un error debido a que **y** es 0 justo antes del cálculo del cociente y el resto. Nos lleva 4 horas encontrar y corregir el error en el cálculo de **y**. Luego, más adelante, tardamos un día en seguirle la pista a un error que no fue detectado en las aserciones, y determinamos que el cálculo del cociente y el resto fue:

x = 6, y = 3, q = 1, r = 3

Como vemos, las aserciones son verdaderas para estos valores. El problema se debe a que el resto ha debido ser menor estricto que el divisor y no lo es. Detectamos que la condición de continuación del proceso iterativo (el **mientras**) ha debido ser ($r \geq y$) en lugar de ($r > y$). Si hubiésemos sido lo suficientemente cuidadosos en colocar en la aserción del resultado otras propiedades que debe cumplir el cociente y el resto, es decir, si hubiésemos utilizado la aserción más *fuerte* $(x = y*q + r) \wedge (r < y)$, nos habríamos ahorrado un día de trabajo. Corregimos el error insertando la aserción más fuerte:

```
  ...
{ y > 0 }
r := x;  q:=0;
mientras r  ≥ y hacer
        comienzo  r :=  r - y;  q := q+1  fin;
{  (x = y*q + r)∧(r < y) }
...
```

Las cosas marchan bien por un tiempo, pero un día obtenemos una salida incomprensible. Resulta que el algoritmo del cociente y resto produjo un resto negativo r = -2. Y nos damos cuenta que r es negativo debido a que inicialmente x era –2. Hubo otro error en el cálculo de los datos de entrada del algoritmo de cociente y resto, pues se supone que x no puede ser

negativo. Hemos podido ahorrar tiempo en detectar este error si nuevamente hubiésemos sido cuidadosos en fortalecer suficientemente las aserciones. Una vez más , arreglamos el error y reforzamos las aserciones:

```
  ...
{ ( x ≥ 0) ∧ (y > 0) }
r := x;  q:=0;
mientras r  ≥ y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
{  (x = y*q + r)∧( 0 ≤ r < y) }
...
```

Parte del problema que hemos confrontado se debe a que no hemos sido lo suficientemente cuidadosos en especificar lo que el trozo de programa debe hacer. Hemos debido comenzar por escribir la aserción inicial (x ≥ 0) ∧ (y > 0) y la aserción final (x = y*q + r)∧(0 ≤ r < y) antes de escribir el trozo de programa, ya que son estas aserciones las que conforman la definición de cociente y de resto.

Pero, ¿qué podemos decir del error que cometimos en la condición del proceso iterativo?. ¿Lo hemos podido prevenir desde un comienzo?. ¿Existe alguna manera de probar, a partir del programa y las aserciones, que las aserciones son verdaderas en cada punto de la ejecución del programa donde ellas aparecen?. Veamos lo que podemos hacer.

Como antes del mientras se cumple x = r y q = 0, vemos que también se cumple parte del resultado antes del mientras:

(1) x = y*q + r

Además , por las asignaciones que se hacen en el cuerpo del mientras, podemos concluir que si (1) es verdad antes de la ejecución del cuerpo del mientras entonces es verdad después de su ejecución. Así (1) será verdad antes y después de la ejecución de *cada* iteración del mientras. Insertemos (1) en los lugares que corresponde y hagamos las aserciones tan fuertes como sea posible:

```
  ...
{ ( x ≥ 0) ∧ (y > 0) }
r := x;  q:=0;
{ (0 ≤ r) ∧ (0 < y) ∧ (x = y*q + r) }
mientras r ≥ y hacer
        comienzo
        { (0 ≤ r) ∧ (0 < y ≤ r) ∧ (x = y*q + r) }
        r := r-y;  q := q+1
        { (0 ≤ r) ∧ (0 < y) ∧ (x = y*q + r) }
        fin;
{  (x = y*q + r) ∧ (0 ≤ r < y) }
...
```

¿Cómo podemos probar que la condición es correcta?. Cuando el mientras termina la condición debe ser falsa y queremos que $r < y$, por lo que el complemento, $r \geq y$ debe ser la condición correcta del mientras.

Del ejemplo anterior se desprende que si supiéramos encontrar las aserciones más fuertes posibles y aprendiéramos a razonar cuidadosamente sobre aserciones y programas, no cometeríamos tantos errores, sabríamos que nuestro programa es correcto y no necesitaríamos depurar programas (sólo correríamos casos de prueba para aumentar la confianza en un programa que sabemos está correcto; encontrar un error sería la excepción y no la regla). Por lo que los días gastados en correr casos de prueba, examinar listados y buscar errores, podrían ser invertidos en otras actividades.

Tercer ejemplo (razonar en términos de propiedades del problema y no por casos de prueba):

Para mostrar lo efectivo que puede ser adoptar una metodología como la que proponemos, ilustremos el siguiente ejemplo:

Supongamos que tenemos un recipiente de café que contiene algunos granos negros y otros blancos y que el siguiente proceso se repite tantas veces como sea posible:

Se toma al azar dos granos del recipiente. Si tienen el mismo color se botan fuera y colocamos otro grano negro en el recipiente (hay suficientes granos negros disponibles para hacer esto). Si tienen color diferente, se coloca el grano blanco de vuelta en el recipiente y se bota el grano negro.

La ejecución de este proceso reduce en uno el número de granos en el recipiente. La repetición de este proceso terminará con un grano en el recipiente ya que no podremos tomar dos granos para continuar el proceso. La pregunta es: ¿qué puede decirse del color del grano final en función del número de granos blancos y negros inicialmente en el recipiente? Piense un poco....

Como vemos, no ayuda mucho intentar con casos de prueba. No ayuda mucho ver qué pasa cuando hay dos granos de color distinto, o dos blancos y uno negro, etc.; aunque examinar casos pequeños permite obtener una mejor comprensión del problema.

En lugar de esto, proceda como sigue: quizás exista una *propiedad sencilla de los granos en el recipiente que permanece cierta una vez que tomamos dos granos y colocamos el grano que corresponde en el recipiente,* y que junto con el hecho de que sólo quedará un grano, pueda dar la respuesta. Como la propiedad siempre permanecerá cierta, nosotros la llamamos *un invariante*.

Tratemos de buscar tal propiedad. Suponga que cuando terminamos tenemos un grano negro. ¿Qué propiedad es verdad cuando terminamos que podríamos generalizar, quizás, para que sea nuestro invariante? Una sería “existe un número impar de granos negros en el recipiente”, así, quizás la propiedad de permanecer impar el número de granos negros sea

cierta. Sin embargo, este no es el caso, pues el número de granos negros cambia de par a impar o de impar a par con cada ejecución de un paso del proceso. Pero también habría cero granos blancos cuando terminamos; es posible que la propiedad de permanecer par el número de granos blancos sea cierta. Y, en efecto, cada ejecución del proceso termina con dos granos blancos de menos en el recipiente o con la misma cantidad de granos blancos. Por lo tanto, el último grano es negro si inicialmente hay un número par de granos blancos; de otra forma, el último grano será blanco.

La solución al problema anterior es sumamente simple, pero extremadamente difícil de encontrar viendo sólo casos de prueba. El problema es más fácil de resolver si buscamos propiedades que permanecen ciertas. Una vez encontradas, estas propiedades nos permiten ver de una manera sencilla que hemos encontrado una solución. Este problema nos permite ver también el siguiente principio: hay que conocer y entender en detalle el enunciado del problema y las propiedades de los objetos que van a ser manipulados por el programa que resuelve el problema.

2. Nociones sobre Teoría de Conjuntos y Lógica

Para llevar a cabo nuestro propósito de especificar formalmente los problemas y demostrar rigurosamente la correctitud de nuestro programas, introduciremos algunas nociones sobre Teoría de Conjuntos y Lógica. Los cconceptos y notación dados nos permitirán unificar la notación que utilizaremos en las especificaciones y las demostraciones de correctitud de programas.

2.1. Conjuntos, Relaciones, Funciones

Un *conjunto* es una colección de objetos distintos, o elementos como normalmente son llamados. Por ejemplo: el conjunto {3,5} consiste de los enteros 3 y 5 y es el mismo conjunto que {5,3}.

{} representa al conjunto vacío, es decir, aquel que no contiene elementos. También se representa por $\varnothing$.

La notación anterior describe a un conjunto de manera explícita o por extensión, listando todos sus elementos. También podemos describir un conjunto de manera implícita o por compresión:

$$\{i: \text{existe un natural j tal que } i= 2^j\}$$

La *cardinalidad o tamaño* de un conjunto es el número de elementos del conjunto, y se denota por $|A|$ o card(A). Así card({3,5})=2

Luego definiremos más formalmente lo que es una operación, sin embargo damos algunas operaciones que todos conocemos sobre conjuntos:

Operaciones entre conjuntos: Unión, Intersección, Diferencia.
Operaciones que producen un valor booleano (verdadero o falso): la igualdad entre conjuntos, la pertenencia ($a \in A$), la no pertenencia ($a \notin A$), la inclusión ($A \subseteq B$)
Operaciones que producen un número: Si A es un conjunto de números entonces *min* A es el menor elemento de A y *max* A es el mayor elementos de A.

El *conjunto de partes de un conjunto* A, detonado por 2^A, es el conjunto cuyos elementos son todos los subconjuntos de A, $2^A=\{ B : B \subseteq A \}$. Una *partición de un conjunto* A es un conjunto cualquiera $P =\{ A_1, A_2, \ldots , A_n \}$tal que para todo i entre 1 y n se cumple que A_i es distinto de $\varnothing$, $A_i \subseteq A$ y cualesquiera dos elementos de P, A_i y A_j, con i distinto de j, tienen intersección vacía y la unión de todos es el conjunto A.

Relaciones y Funciones

Sean A y B dos conjuntos. El *producto cartesiano* de A y B, denotado por AxB, es el conjunto de los pares ordenados (a,b) donde $a \in A$ y $b \in B$:

$$AxB=\{(a,b): a \in A \text{ y } b \in B\}$$

La *cardinalidad* de AxB es $|A|*|B|$

Una *relación binaria* en los conjuntos A y B es un subconjunto de AxB. Si R es una relación y (a,b) $\in$ R, entonces también escribimos a R b. El *dominio de R*, denotado dom(R), es el conjunto de los elementos a de A para los cuales existe un par (a,b) en R, el *rango de R*, denotado rango(R), es el conjunto de los elementos b de B para los cuales existe un par (a,b) en R.

Por ejemplo, sea P el conjunto de los seres humanos. Una relación en PxP es la relación progenitor:

Progenitor= {(a,b): a $\in$ P y b $\in$ P y a es progenitor de b}

Sea N el conjunto de los números naturales. Una relación en NxN es la relación sucesor:

Sucesor={(i, i+1): i $\in$ N}

La relación identidad en AxA, denotada por I, es la relación:

I = {(a,a): a $\in$ A}

Una relación en AxB puede que no contenga pares (a,b) para algún a en A, en este caso decimos que la relación es *parcial*. Por otro lado, decimos que la relación es *total* si existe al menos un par (a,b) para cada a en A. Si para cada b en B existe un par (a,b) en la relación decimos que R es *sobre* B.

La *relación secuenciación* de dos relaciones R en AxB y S en BxC, denotada por R.S, es la relación en AxC definida por:

a R . S c si y sólo si Existe b tal que a R b y b S c

Por ejemplo, la relación (Progenitor . Progenitor) es la relación Abuelo={(a,c): a es abuelo de c}

La secuenciación es *asociativa*: (R.S).T = R.(S.T) por lo que podemos omitir los paréntesis y escribir R.S.T. R^i denota la relación R secuenciada con ella misma i-veces, cuando R es una relación en AxA. R^0 representa a la relación identidad.

De igual forma la *relación inversa* de una relación R en AxB es una relación, denotada por R^{-1} en BxA y definida por {(b,a): (a,b) $\in$ R}

Una relación R en AxA es *reflexiva* si I $\subseteq$ R, es *simétrica* si siempre que el par (a,b) esté en R se tiene que el par (b,a) está en R, es *transitiva* si siempre que el par (a,b) esté en R y el par (b,c) esté en R se tiene que el par (a,c) está en R.

Una función f de A en B es una relación en AxB donde para cada a $\in$ A existe a lo sumo un par (a,b) y se denota por:

$$f : A \rightarrow B$$

Por ser una relación, a una función aplican todos los conceptos dados sobre relaciones. Cada par de una función lo podemos escribir de la forma (a, f(a)), f(a) es llamado el valor de la función f para el argumento a. Diremos que la función f no está definida para un elemento a de A si f no contiene el par (a,b) cualquiera sea b en B.

La relación sucesor es una función. Así sucesor(1)=2, sucesor(i)=i+1.

Una *operación n-aria* es una función del producto cartesiano de n conjuntos en un conjunto, decimos que la operación es de *aridad* n. Por ejemplo, si P(A) representa el conjunto de todos los subconjuntos de un conjunto A, entonces la función que asigna a cada par X, Y de elementos de P(A), la unión de X e Y es una operación binaria de P(A)xP(A) en P(A), el valor de la operación unión aplicada a X e Y la denotamos usualmente por X$\cup$Y, y decimos que $\cup$ es el *operador* unión. Decimos que X e Y son los operandos de la operación unión.

Secuencias, números enteros y reales

Una *secuencia* de largo n de elementos en un conjunto B, es una función total de A={0, 1, 2, ... , n-1} en B, donde B es cualquier conjunto y n cualquier número natural. Por ejemplo la función S={(0,b_0), (1,b_1), (2,b_2), ... , (n-1,b_{n-1})} es una secuencia y decimos que es de largo n, denotamos el largo de una secuencia s por *largo(s) o por* $/s/$, y vemos que en general el largo de una secuencia es igual a la cardinalidad de su dominio.

Hay otra forma de denotar una secuencia, por ejemplo, la secuencia S anterior también la podemos denotar de la siguiente forma: s = <b_0, b_1, ... , b_{n-1}>. Decimos que el primer elemento de la secuencia es b_0, el segundo elemento es b_1, y en general el elemento i-ésimo de la secuencia es b_{i-1}. También denotaremos los elementos de una secuencia en la forma s[i], donde s[0] es el primer elemento, etc. Cuando el dominio de una secuencia sea vacío diremos que la secuencia es vacía y la denotamos por < >.

Una operación binaria entre secuencias es la *concatenación*, denotada por $||$. Si B=<b_0, b_1, ... , b_{n-1}> y C = <c_0, c_1, ... , c_{m-1}> son dos secuencias entonces B $||$ C es la secuencia <b_0, b_1, ... , b_{n-1}, c_0, c_1, ... , c_{m-1}> la cual tiene largo n+m. Note que el elemento n+1 de esta secuencia es c_0, el elemento n+2 es c_1, etc.

La operación *primero* aplicada a una secuencia nos dá el primer elemento de la secuencia, así primero(<1,2,3>)=1. La operación *resto* aplicada a una secuencia s de largo n nos devuelve la secuencia de largo n-1, cuyo i-ésimo elemento, 1$\leq$ i $\leq$n, es el (i+1)-ésimo elemento de s, así resto(<1,2,3>)=<2,3>. Note que las operaciones primero y resto no están definidas para la secuencia vacía.

Números

Normalmente utilizaremos los siguientes conjuntos:

El conjunto de los números enteros Z: {..., -2, -1, 0, 1, 2, ...}
El conjunto de los números naturales N: {0, 1, 2, 3, ...}
El conjunto de los números reales que denotaremos por $\Re$.

Las siguientes operaciones toman por operandos números enteros o reales:

+, -, *	suma, resta, multiplicación
/	división; produce un número real
<, >, =, ≥, ≤, ≠	operadores de comparación; producen verdadero o falso
abs(x) ó $\lvert x \rvert$	valor absoluto
$\lceil x \rceil$	menor entero mayor o igual que x
$\lfloor x \rfloor$	mayor entero menor o igual que x (parte entera de x)

Las siguientes operaciones, **div** y **mod**, tienen sólo enteros como operandos:

Dados dos números enteros $\mathbf{x} \geq 0$, $\mathbf{y} > 0$, el cociente y resto de la división entera de **x** entre **y** son los números enteros **q** y **r**, respectivamente, que satisfacen:

$$\mathbf{x} = \mathbf{y} * \mathbf{q} + \mathbf{r}, \quad 0 \leq \mathbf{r} < \mathbf{y}$$

x div **y**	denotará el cociente de la división entera de **x** entre **y**.
x mod **y**	denotará el resto de la división entera de **x** entre **y**.
	Note que **x** mod **y = x – y * (x** div **y)**

Ejercicios: Guías de Estructuras Discretas I. Autor: Pedro Borges.

1) Capítulo 2, ejercicios 1, 2, 3, 4, 12 con lógica, 13ª con lógica.
2) Capítulo 4: 4, 13.
3) Capítulo 5: 5

2.2. Proposiciones y Predicados

Proposiciones

Una proposición es una expresión sobre un conjunto de *variables lógicas o booleanas*, es decir, variables que sólo pueden tomar los valores "verdadero" ó "falso" (denotaremos "verdadero" por V, y "falso" por F). Estas expresiones se definen como sigue:

1) V y F son proposiciones.
2) Una variable lógica (o identificador) es una proposición.
3) Si a es una proposición entonces ($\neg$ a) es una proposición.
4) Si a y b son proposiciones entonces así lo son ($a \wedge b$), ($a \vee b$), ($a \Rightarrow b$), ($a \equiv b$)

Ejemplo de proposiciones: V, (($\neg$ a)$\vee$ (a$\wedge$b)), ((a$\vee$b)$\vee$(F$\vee$x))

Como vemos en la sintaxis anterior, se han definido cinco operadores sobre valores de tipo booleano, uno de aridad 1 y cuatro de aridad 2 (binarios):

Negación:	(no a), o ($\neg$ a)
Conjunción:	(a y b), o (a$\wedge$b)
Disyunción:	(a o b), o (a$\vee$b)
Implicación:	(a implica b), o (a $\Rightarrow$ b), b es el antecedente y c la consecuencia
Equivalencia:	(a es equivalente a b), o (a $\equiv$ b)

Evaluación de proposiciones.

Las operaciones de negación, conjunción, disyunción, implicación y equivalencia se definen de manera natural (desde el punto de vista lógico o de sentido común), como sigue:

a	b	$\neg$ a	(a$\wedge$b)	(a$\vee$b)	(a $\Rightarrow$ b)	(a $\equiv$ b)
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

Podemos Evaluar expresiones booleanas de acuerdo a esta tabla. Por ejemplo: evaluemos la proposición (($a \wedge (b \Rightarrow c)) \vee \neg c$) para a=F, b=V, c=F. Reemplazamos en la expresión los valores de verdad de las variables y obtenemos (($F \wedge (V \Rightarrow F)) \vee \neg F) = V$.

Podemos omitir paréntesis de acuerdo a las siguientes reglas de precedencia de operadores:

a) Secuencias del mismo operador son evaluadas de derecha a izquierda (ver David Gries). Por ejemplo: $a \Rightarrow b \Rightarrow c \Rightarrow d$ es lo mismo que $(a \Rightarrow (b \Rightarrow (c \Rightarrow d)))$. Como veremos más adelante, los otros tres operadores binarios ($\wedge$, $\vee$, $\equiv$) son asociativos, por lo que no importará por donde comience la evaluación.

b) El orden de evaluación de operadores consecutivos diferentes es:
   - negación (prioridad más alta),
   - conjunción y disyunción, la misma prioridad,
   - implicación y equivalencia, la misma prioridad.

   Por ejemplo: $b \Rightarrow c \Rightarrow d \wedge e$ es lo mismo que $b \Rightarrow (c \Rightarrow (d \wedge e))$
   $a \vee b \wedge c$ no sabremos cómo evaluarla.

Podemos convertir expresiones en español a forma de proposicional:

Por ejemplo la frase “Si llueve y me quedo en casa no me mojaré” la podemos convertir en una proposición de la siguiente forma:

Buscamos, si se puede, algunas proposiciones atómicas:

Llueve: a
Me quedo en casa: b
Me mojo: c

Y la frase en español quedaría en la forma: $(a \wedge b) \Rightarrow \neg c$

Ejercicios: pag. 17 Gries, números 1, 2, 3, 4.

Una *Tautología* es una proposición que es verdad cualesquiera sean los valores de verdad de las variables que intervienen en ella. Por ejemplo $(a \vee \neg a)$ es una tautología.

La lista siguiente son tautologías:

Leyes Conmutativas:

$(E1 \wedge E2) \equiv (E2 \wedge E1)$
$(E1 \vee E2) \equiv (E2 \vee E1)$
$(E1 \equiv E2) \equiv (E2 \equiv E1)$

Leyes Asociativas:

$((E1 \wedge E2) \wedge E3)) \equiv (E1 \wedge (E2 \wedge E3))$
$((E1 \vee E2) \vee E3)) \equiv (E1 \vee (E2 \vee E3))$
$((E1 \equiv E2) \equiv E3)) \equiv (E1 \equiv (E2 \equiv E3))$

Leyes distributivas:

$(E1 \vee (E2 \wedge E3)) \equiv ((E1 \vee E2) \wedge (E1 \vee E3))$
$(E1 \wedge (E2 \vee E3)) \equiv ((E1 \wedge E2) \vee (E1 \wedge E3))$

Leyes de De Morgan

$\neg(E1 \wedge E2) \equiv (\neg E1 \vee \neg E2)$
$\neg(E1 \vee E2) \equiv (\neg E1 \wedge \neg E2)$

Ley de Doble Negación:

$\neg(\neg E1) \equiv E1$

Ley del Tercero Excluido:

$E1 \vee \neg E1 \equiv V$

Ley de la Contradicción:

$E1 \wedge \neg E1 \equiv F$

Ley de la Implicación:

$(E1 \Rightarrow E2) \equiv \neg E1 \vee E2$

Ley de la Equivalencia:

$(E1 \equiv E2) \equiv (E1 \Rightarrow E2) \wedge (E2 \Rightarrow E1)$

Leyes de simplificación de la disyunción:

$E1 \vee E1 \equiv E1$
$E1 \vee V \equiv V$
$E1 \vee F \equiv E1$
$E1 \vee (E1 \wedge E2) \equiv E1$

Leyes de simplificación de la conjunción:

$E1 \wedge E1 \equiv E1$
$E1 \wedge V \equiv E1$
$E1 \wedge F \equiv F$
$E1 \wedge (E1 \vee E2) \equiv E1$

Ley de identidad:

$E1 \equiv E1$

Regla de Sustitución: Si e1≡e2 es una tautología y E(p) es una proposición donde aparece la variable p, entonces E(e1) ≡ E(e2) y E(e2) ≡ E(e1) son tautologías.

Regla de Transitividad: Si e1≡e2 y e2≡e3 son tautologías entonces e1≡e3 es una tautología.

Ejemplo de demostración de tautologías utilizando las leyes y reglas:

Demostrar que $(b \Rightarrow c) \equiv (\neg c \Rightarrow \neg b)$ es una tautología

$(b \Rightarrow c)$
$\equiv \neg b \vee c$ (por la ley de implicación)
$\equiv c \vee \neg b$ (por la leyconmutativa)
$\equiv \neg\neg c \vee \neg b$ (por la ley de negación y regla de sustitución)
$\equiv \neg c \Rightarrow \neg b$ (por la ley de implicación)

Así $(b \Rightarrow c) \equiv (\neg c \Rightarrow \neg b)$ es una tautología por la regla de transitividad.

Los pasos de la demostración deben leerse como sigue: partiendo del hecho que $(b \Rightarrow c)$ es verdadero (lo cual es una suposición) se tiene que $\neg b \vee c$ es verdadero pues $(b \Rightarrow c) \equiv \neg b \vee c$ es una tautología (si $V \equiv x$ es verdadero entonces x debe ser verdadero), etc.

Ejercicios: pag. 26 y 27 del Gries, números 1, 3, 4, 5, 6(b,e)

Predicados:

Una proposición, como $\mathbf{p} \vee (\mathbf{q} \wedge \mathbf{s})$, la podemos ver como una función de $\{V,F\}^3$ en $\{V,F\}$. Por ejemplo para **p**=V, **q**=F y **s**=V el valor de la proposición es V. (V,F,V) se denomina *estado* de las variables **p**, **q** y **s**. Para cada estado tendremos un valor de verdad para la proposición.

Ahora queremos extender la noción de estado para permitir a las variables contener valores de otros tipos, como enteros, conjuntos, etc. Con este fin en mente, la noción de una proposición será generalizada de dos maneras, dando lugar a los *predicados*:

1) En una proposición, una variable booleana podrá ser reemplazada por cualquier expresión que tenga valor V ó F, por ejemplo: $\mathbf{x} > \mathbf{y}$.

2) Definiremos los cuantificadores existencial ($\exists$) y universal ($\forall$), lo cual conllevará a explicar las nociones de identificadores libres y ligados y al alcance de las variables en las expresiones.

Consideremos la expresión $(\mathbf{x} > \mathbf{y})$, donde **x** e **y** son de tipo entero. Cuando evaluamos la expresión para valores particulares de **x** e **y**, obtenemos un valor V ó F. Por lo tanto esta expresión puede reemplazar cualquier identificador en una proposición. Por ejemplo, reemplazando **p** en $\mathbf{p} \vee (\mathbf{q} \wedge \mathbf{s})$ por $(\mathbf{x} > \mathbf{y})$, obtenemos:

$$(\mathbf{x} > \mathbf{y}) \vee (\mathbf{q} \wedge \mathbf{s})$$

La expresión que resulta de reemplazar una variable booleana en una proposición por una expresión que al evaluarla dé V ó F, se denomina *predicado*. Note que una proposición es un caso particular de predicado. Otros predicados válidos son:

$((\mathbf{x} > \mathbf{y}) \wedge (\mathbf{z} < \mathbf{y})) \vee (\mathbf{x}+\mathbf{y} \geq \mathbf{z})$
$(\mathbf{x} > \mathbf{y} \wedge \mathbf{z} < \mathbf{y}) \vee \mathbf{x}+\mathbf{y} \geq \mathbf{z}$

La segunda expresión muestra que tantos paréntesis no serán necesarios, pues consideraremos que los operadores lógicos tendrán prioridad más baja que los operadores relacionales (como <) y los aritméticos.

Evaluación de predicados:

La evaluación de un predicado es similar a la evaluación de una proposición. Todas las variables son reemplazadas por sus valores (que llamamos el estado de las variables), estos valores serán de los tipos a los que corresponda cada variable (entero, lógico, etc.). Luego se procede a evaluar la expresión resultante de reemplazar las variables por sus valores. Por lo que un predicado lo podemos ver como una función cuyo conjunto de partida es el producto cartesiano de los conjuntos de valores de las variables y el conjunto de llegada es {V, F}. Por ejemplo: el predicado $\mathbf{x} > \mathbf{y} \vee (\mathbf{q} \wedge \mathbf{s})$ en el estado x = 3, y = 4, q = F, s = F, tiene el valor de $3 > 4 \vee (F \wedge F) = F$

Ejercicios: 1, 2, 3(g,h,i,j) pag. 70 de Gries.

Cuantificadores Existencial y Universal

Cuantificador Existencial:

Sean m y n dos enteros con $m \leq n$. Consideremos el predicado:

$$(1) \qquad E_m \vee E_{m+1} \vee \ldots \vee E_{n-1}$$

donde cada E_i es un predicado. (1) es verdad en cada estado en el cual al menos uno de los E_i es verdad, y falso en caso contrario. Podemos expresar este hecho utilizando el cuantificador existencial $\exists$ de la siguiente forma:

$$(2) \qquad (\exists i: m \leq i < n: E_i)$$

Los valores i que satisfacen $m \leq i < n$ se llama *el rango* del *identificador cuantificado i*. (2) se lee como sigue: "Existe al menos un entero i tal que i está entre m y (n-1) inclusive, para el cual se cumple E_i"

Ejemplo: $(\exists i: m \leq i < n: x * i > 0)$, y se lee: existe al menos un entero i entre m y n-1 tal que x*i es mayor que 0

Por la definición anterior se tiene que:

$(\exists i: m \leq i < m: E_i) \equiv F$ (la disyunción de cero predicados es falso)

Para $k \geq m$, $(\exists i: m \leq i < k+1: E_i) \equiv (\exists i: m \leq i < k: E_i) \vee E_k$

Cuantificador Universal:

Sean m y n dos enteros con $m \leq n$. Consideremos el predicado:

$$(3) \qquad E_m \wedge E_{m+1} \wedge \ldots \wedge E_{n-1}$$

donde cada E_i es un predicado. (3) es verdad en cada estado en el cual todos los E_i son verdad para ese estado, y falso en caso contrario. Podemos expresar este hecho utilizando el cuantificador universal $\forall$ de la siguiente forma:

$$(4) \qquad (\forall i: m \leq i < n: E_i)$$

El conjunto de valores que satisfacen $m \leq i < n$ se llama *el rango* del *identificador cuantificado i*. (4) se lee como sigue: "Para todo entero i tal que i está entre m y (n-1) inclusive, se cumple E_i"

Ejemplo: $(\forall i: m \leq i < n: x * i > 0)$, y se lee: para todo i entre m y n-1 se tiene que x*i es mayor que 0

Por la definición anterior se tiene que:

$(\forall i: m \leq i < m: E_i) \equiv V$ (la conjunción de cero predicados es verdadero)
Para $k \geq m$: $\forall i: m \leq i < k+1: E_i) \equiv (\forall i: m \leq i < k: E_i) \wedge E_k$

Podemos definir el cuantificador universal en términos del cuantificador existencial. Por ejemplo, para m = 3 y n = 8:

$(\forall i: 3 \leq i < 8: E_i)$
$\equiv E_3 \wedge E_4 \wedge E_5 \wedge E_6 \wedge E_7$ (por definición de la notación $\forall$...)
$\equiv \neg\neg(E_3 \wedge E_4 \wedge E_5 \wedge E_6 \wedge E_7)$ (ley de negación)
$\equiv \neg(\neg E_3 \vee \neg E_4 \vee \neg E_5 \vee \neg E_6 \vee \neg E_7)$ (ley de De Morgan)
$\equiv \neg(\exists i: 3 \leq i < 8: \neg E_i)$ (definición del existencial)

Ejemplo de uso de los cuantificadores:

Existe un teorema matemático que dice que existen números primos arbitrariamente grandes. Este teorema lo podemos expresar más formalmente como sigue:

$(\forall n: 0 < n: (\exists i : n < i: primo(i)))$ implícitamente n e i son enteros, podría ser necesario explicitarlo en el rango!!.

donde $primo(i) = (1 < i \wedge (\forall j : 1<j<i: i \bmod j \neq 0))$

Ejercicios: pag. 75 Gries, número 6.

Podemos cuantificar sobre otros rangos que no sean los números enteros. Por ejemplo, si P(p) representa la expresión "p es una persona" y M(x) representa la expresión "x es mortal" entonces la expresión "todas las personas son mortales" se puede expresar por $(\forall p: P(p): M(p))$.

Supongamos que hemos probado que el predicado max (n, -n) = abs (n), donde max es la función máximo entre dos números y abs la función valor absoluto, es una tautología en los

números enteros, es decir, en cada estado de la variable n la igualdad es verdadera. Por lo tanto es verdadero para todo entero n, por lo que el predicado siguiente es verdadero:

$(\forall n : n \in Z: \max(n, -n) = abs(n))$

si en el contexto podemos sobreentender que n está en los enteros, podemos escribir:

$(\forall n : \max(n, -n) = abs(n))$, es decir, no especificar el rango.

Cualquier tautología E es equivalente al mismo predicado E pero con todas sus variables libres $i_1, i_2, \ldots, i_m$ cuantificadas universalmente, es decir, es equivalente a $(\forall\ i_1, i_2, \ldots, i_m : E)$. Por ejemplo: $m \leq 0 \vee m > 0$ es equivalente a $\forall m\ (m \leq 0 \vee m > 0)$

Variables libres y ligadas:

El predicado:

$$(5) \qquad (\forall i: m \leq i < n: x * i > 0)$$

afirma que x multiplicado por cualquier entero entre m y n-1 (inclusive) es mayor que cero. Vemos que esto es verdad si y sólo si x y m son mayores que cero o x es menor que cero y n es a lo sumo 0. Por lo tanto (5) es equivalente a:

$$(6) \qquad (x > 0 \wedge m > 0) \vee (x < 0 \wedge n \leq 0)$$

(cuando decimos (5) es equivalente a (6) queremos decir que (5) ≡ (6) es una tautología). Por lo tanto el valor de verdad de (5) depende del estado de las variables x, m, n, pero no de i. más aún, el significado de (5) no cambia si reemplazamos i por j. Diremos que x, n, m son *variables libres* del predicado (5) (ó (6)), y la variable i es una *variable ligada* en (5), y ella está ligada al cuantificador ∀.

Ejercicios: pag. 79 Gries.

Otros cuantificadores:

Sea X un conjunto y $\oplus$ una operación binaria sobre X conmutativa, asociativa y con elemento neutro e, es decir para todo x, y, z en X:

$x \oplus y = y \oplus x$
$x \oplus (y \oplus z) = (x \oplus y) \oplus z$
$e \oplus x = x \oplus e$

Para la secuencia x(i), $0 \leq i$, y n un número natural, la expresión $x(0) \oplus x(1) \oplus \ldots \oplus x(n-1)$ la escribimos como $(\oplus\ i : 0 \leq i < n: x(i))$.

Y tenemos:

$(\oplus\ i : 0 \le i < 0 : x(i)) = e$
$(\oplus\ i : 0 \le i < n+1 : x(i)) = (\oplus\ i : 0 \le i < n : x(i)) \oplus\ x(n)$

La última línea pude ser acompañada con la nota "separación de i=n".

En general, esta cuantificación es de la forma: $(\oplus\ x : R(x) : F(x))$, donde x es una lista de variables, R es un predicado con variables libres x, llamado el *rango de la cuantificación*, y F es llamado el *término*.

Tenemos entonces que $(\oplus\ x : falso : F(x)) = e$, el elemento neutro de $\oplus$.

Los siguientes operadores son conmutativos, asociativos y poseen elemento neutro: la suma +, la multiplicación, el máximo max, el mínimo min, la conjunción $\wedge$, la disyunción $\vee$. El elemento neutro de +, *, max, min, $\wedge$, $\vee$ son respectivamente 0, 1, $-\infty$, $+\infty$, verdadero, falso.

Utilizamos el símbolo $\sum$ como símbolo de cuantificación de la suma de n números x(0) + x(1) + ... + x(n-1). Colocamos $(\sum\ i : 0 \le i < n : x(i))$ en lugar de $(+\ i : 0 \le i < n : x(i))$. De igual forma usamos $\prod$ para denotar la cuantificación de la multiplicación en lugar de *: $(\prod\ i : 0 \le i < n : x(i))$. Como ya vimos, utilizamos $\forall$ y $\exists$ para cuantificar respectivamente a $\wedge$ y $\vee$.

Ejemplos:
$(\sum\ i : 3 \le i < 5 : i^2) = 3^2 + 4^2 = 25$
$(\sum\ x, y : 0 \le x < 3\ \wedge\ 0 \le y < 3\ : x*y) = 9$
$(\prod\ i : falso : x(i)) = 1$
$(max\ i : 3 \le i < 5 : i^2) = 16$

Se tienen las siguientes propiedades:

a) max y min son distributivos uno respecto al otro:
$x\ min\ (max\ i : R : F(i)) = (max\ i : R : x\ min\ F(i))$
$x\ max\ (min\ i : R : F(i)) = (min\ i : R : x\ max\ F(i))$

b) Si R no es vacío:
$x + (max\ i : R : F(i)) = (max\ i : R : x+F(i))$
$x + (min\ i : R : F(i)) = (min\ i : R : x+F(i))$

c) $(max\ i : R \vee S : F(i)) = (max\ i : R : F(i))\ max\ (max\ i : S : F(i))$, igual para min.

Otro cuantificador es el *cuantificador de conteo*, denotado por #. La función #: {falso, verdadero} $\rightarrow$ {0, 1}, se define como #(falso) = 0 y #(verdadero) = 1. La expresión (# i: R(i) : F(i)), donde F(i) es un predicado, se define como $(\sum\ i : R(i) : \#(F(i)))$, y representa el número de valores en el rango R para los cuales F es verdadero. En particular, (# i: falso : F(i)) = 0

Ejercicios: Página 49 Kaldewaij.

Tautologías y Predicados más fuertes que otros:

Un predicado P es una tautología si para cualquier estado este es verdadero, la expresión "P es una tautología" la denotamos por [P]. Ejemplo, el predicado $x > 0 \Rightarrow x \geq 0$ es una tautología y denotamos este hecho de la forma: $[\, x > 0 \Rightarrow x \geq 0 \,]$.

Note que $[\, x \geq 0 \Rightarrow x > 0 \,]$ no se cumple, es decir $x \geq 0 \Rightarrow x > 0$ no es una tautología.

Si $[\, P \Rightarrow Q \,]$ entonces diremos que *P es más fuerte que Q* ó que *Q es más débil que P*. Ejemplo, como $[\, x > 0 \Rightarrow x \geq 0 \,]$ se cumple decimos que $x > 0$ es más fuerte que $x \geq 0$. Se usa la expresión más *fuerte* en el sentido que el conjunto de estados que hacen verdad al antecedente de la implicación es un subconjunto (son menos estados) del conjunto de estados que hacen verdad al consecuente de la implicación.

Cuando en español decimos "el predicado Q se cumple **si** se cumple el predicado P", queremos decir que $P \Rightarrow Q$ es una tautología. De igual forma, cuando decimos "Q se cumple **sólo si** se cumple P", queremos decir $Q \Rightarrow P$ es una tautología. Por lo tanto cuando decimos "P se cumple **si y sólo si** se cumple Q", queremos expresar que $P \equiv Q$ es una tautología.

Las leyes y reglas vistas para proposiciones (ley conmutativa, de De Morgan, etc.) se extienden de manera natural a predicados.

3. Proceso de desarrollo de una solución algorítmica de un problema

Estas notas se centran en la enseñanza de la programación "en pequeño". La habilidad para hacer programas pequeños es necesaria para desarrollar programas grandes, aunque puede no ser suficiente, pues el desarrollo de programas grandes requiere de otras técnicas que no serán tratadas en estas notas.

Los pasos generales para resolver un problema mediante un algoritmo son los siguientes:

a) Especificación del problema.
b) Diseño y construcción del algoritmo.

3.1. Especificación del problema

La especificación de un problema consiste antes que nada en precisar la información del problema, la cual la constituyen los datos de entrada y los resultados que se desea obtener, y formular las relaciones existentes entre los datos de entrada y los resultados. Para ello será necesario:

- identificar los objetos involucrados y darles un nombre (una variable que los identifique).
- reconocer la clase o tipo de cada uno de los objetos, lo cual significa conocer el conjunto de valores que los objetos pueden poseer y las operaciones que pueden efectuarse sobre estos.
- Describir lo que se desea obtener en términos de relaciones entre los objetos involucrados.

Dado un enunciado de un problema, normalmente en lenguaje natural, se busca mejorar el enunciado inicial del problema hasta convertirlo en un enunciado cerrado, es decir, un enunciado completo, no ambiguo, consistente y no redundante. Debemos establecer una *especificación* del problema ayudándonos, para ser lo más precisos que podamos, de lenguajes formales, como el lenguaje de las matemáticas. En este punto interviene el proceso de abstracción que nos permite describir el problema en términos de modelos (por ejemplo, modelos matemáticos; la velocidad de un automóvil la podemos representar por un número real) eliminando la información que no interviene directamente en el resultado esperado (redundante), y especificar el problema en términos de esos modelos, por lo que el problema lo convertimos en un problema equivalente en términos de los modelos.

Como estamos interesados en encontrar un programa (o algoritmo) que resuelva el problema, en lugar de hablar de especificación del problema hablaremos de la *especificación del programa* que resolverá el problema. Esta especificación la haremos en términos funcionales, es decir, describiremos los objetos que representan los datos de "entrada", sus estados y relaciones válidas *requeridas* antes de la ejecución del algoritmo para encontrar la solución del problema. A esta descripción la llamaremos *precondición* del programa. Describiremos los objetos resultado, sus estados y relaciones válidas *deseadas* después de la ejecución del algoritmo, a esta descripción la llamaremos *postcondición*.

Igualmente debemos definir el espacio de estados, el cual vendrá dado por las variables que representarán los objetos que manipulará el programa. *Una especificación funcional* de un programa vendrá dada por el espacio de estados, una precondición (de donde partimos) y una postcondición (a donde queremos llegar). Decimos que la especificación es funcional pues describimos primero los datos de entrada y las condiciones *requeridas* sobre estos datos antes de ejecutar el programa, y luego, los resultados *deseados* que el programa produce.

Ejemplo de especificación:

Problema: Queremos determinar el máximo entre dos números x e y.

Una posible especificación del programa sería:

```
[
   var x,y,z: entero;
  {x=A ∧ y=B}
    máximo
  {z = max(A,B)}
]
```

La primera línea "var x,y,z: entero;" define el espacio de estados. En este caso el espacio de estados es ZxZxZ, donde Z denota el conjunto de los números enteros y está definido por las variables x, y, z, que pueden contener objetos del tipo número entero. Las coordenadas de este espacio corresponden a las variables, la primera coordenada corresponde a x, la segunda a y, la tercera a z. Los elementos del espacio de estados son llamados estados, por ejemplo (1,2,4) y (0,-5,9) son estados. La precondición del programa es el predicado x=A ∧ y=B, el cual establece que justo antes de ejecutar el programa "máximo" el estado es x=A ∧ y=B, z puede contener cualquier valor, por lo que no se menciona explícitamente como en el caso de las variables x e y. El predicado z = max(A,B), es la postcondición del programa "máximo".

A y B se llaman *variables de especificación*, en el sentido que no aparecerán en las instrucciones (o acciones) del programa, sólo sirven para representar valores en las condiciones (pre y post condiciones) y pueden verse en este caso como datos de entrada al programa que pueden ser proporcionados, por ejemplo, mediante la ejecución de una instrucción de lectura. Por los momentos no será relevante conocer cómo se establece la precondición, y en particular, el estado inicial (A,B,?).

Supongamos que tenemos a la mano un programa, llamado "máximo", ya desarrollado, que calcula el máximo entre dos enteros x e y. Diremos que el programa "máximo" cumple la especificación anterior si para todas las ejecuciones de "máximo" comenzando en un estado que cumpla la precondición, este termina (tiene un tiempo de ejecución finito) en un estado que cumple la postcondición.

En general, el hecho que un programa S cumpla una especificación con precondición P y postcondición Q se denota de la siguiente manera, y se conoce con el nombre de **tripleta de Hoare**:

$$\{P\}\ S\ \{Q\}$$

y tiene la siguiente interpretación operacional:

Siempre que una ejecución de S comience en un estado que cumple la precondición P, entonces S terminará (tiene un tiempo de ejecución finito) en un estado que cumple la postcondición Q.

Note que la expresión {P} S {Q} es un predicado, es decir, puede tener un valor verdadero o falso. En efecto, por definición de {P} S {Q}, se sobreentiende que esta proposición está universalmente cuantificada sobre todas las variables de especificación que aparecen en ella, más aún, sobre todas las variables libres que aparecen en ella. En el ejemplo anterior, la especificación:

$$\{x=A \wedge y=B\}\ \text{máximo}\ \{z = \max(A,B)\}$$

tiene el significado siguiente:

$$\forall A \forall B \forall x \forall y \forall z\ (\{x=A \wedge y=B\}\ \text{máximo}\ \{z = \max(A,B)\})$$

que se interpreta de la siguiente manera: para todo número entero A y para todo número entero B, si se tiene que x=A, y=B al comienzo de la ejecución del programa "máximo", entonces "máximo" terminará en un estado que cumple z=max(A,B).

El ejemplo de la sección 1.1. lo podemos especificar como sigue:

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

    { Se tiene que v=V, k=K, l=L, x=X, donde V, K, L, X son números reales no negativos
    y K es distinto de cero. 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 un vehículo al recorrer X kilómetros a la
    velocidad constante V. }
]
```

En este ejemplo el espacio de estados es $\Re \times \Re \times \Re \times \Re \times \Re$ (el producto cartesiano del conjunto de los números reales por el mismo 5 veces). Las coordenadas de este espacio

corresponden a cada una de las variables: la primera coordenada a v, la segunda a k, la tercera a l, la cuarta a x y la quinta a r. V, K, L y X son variables de especificación.

Sobre buenas y malas especificaciones

Considere la siguiente especificación de un programa para calcular el máximo entre dos números enteros cualesquiera:

```
[
  var x,y,z: entero;
 { x=A ∧ y=B }
  máximo
 {z = max(x,y)}
]
```

Cómo la especificación no menciona que x e y no pueden ser modificadas por el programa, éste puede modificar los valores de las variables x, y, z. En efecto, el siguiente programa:

- colocar en x el valor de 1
- colocar en y el valor de 2
- colocar en z el valor de 2

cumple perfectamente con la especificación anterior. Sin embargo, no es éste el programa que queremos ya que este programa no está calculando el máximo entre dos números cualesquiera.

El ejemplo anterior ilustra la utilidad de las variables de especificación como mecanismo para garantizar que cuando termine el programa la variable z contendrá el máximo de los valores *originales* de x e y.

Es bueno mencionar que un principio básico de un buen programador es "el programa no debe hacer ni más ni menos de lo explícitamente requerido". En consecuencia, aunque en ningún momento se menciona explícitamente que x e y no deben cambiar de valor, para evitar la ambiguedad recurrimos a las variables de especificación y obtenemos la especificación:

```
[  var x,y,z: entero;
  {x=A ∧ y=B}
   máximo
  {z = max(A,B)}
]
```

que garantiza que z contendrá el máximo de los valores *originales* de x e y.

Si queremos ir más lejos y explícitamente colocar en la especificación el hecho de que algunas variables no deben ser modificadas por el programa, introducimos la noción de

constante. Utilizamos la palabra **const** en lugar de **var** para expresar el hecho que las variables correspondientes no serán modificadas. Así, para evitar que las variables x e y sean modificadas por el programa máximo, tenemos la especificación:

```
[  const X,Y: entero;
   var z: entero;
   { verdad }
    máximo
   {z = max(X,Y)}
].
```

La precondición "verdad" significa "no se le exige nada al estado inicial de las variables para ejecutar el programa", así todos los estados del espacio de estados cumplen la precondición ("verdad"). En términos de conjuntos podemos razonar como sigue para mostrar que todo estado del espacio de estados satisface la precondición "verdad": Dada la precondición P(x), donde x es una variable sobre el espacio de estados, el conjunto I de estados que satisfacen P(x) es I = { x: x pertenece al espacio de estados $\wedge$ P(x) }. En caso de que P(x) sea la proposición "verdad" (ó V), el conjunto I es igual al espacio de estados en su totalidad (se cumple: [(x pertenece al espacio de estados $\wedge$ V) $\equiv$ x pertenece al espacio de estados]), por lo que cualquier estado del espacio de estados puede ser utilizado como estado inicial para ejecutar el programa (en este caso "máximo").

Utilizaremos letras mayúsculas para denotar a las constantes y a las variables de especificación. En la última especificación el espacio de estados sigue siendo ZxZxZ.

Ejemplos de especificaciones de programas:

1) Problema: Dado un número natural mayor que 1 se quiere determinar todos sus factores primos.

Primero debemos analizar el enunciado y entenderlo bien. El enunciado del problema lo podemos refinar como sigue:

Dado un número natural x mayor que 1 se quiere determinar todos los números naturales z que sean primos y que dividan a x.

Refinando más (introducimos la noción de conjunto):

Dado un número natural x mayor que 1 se quiere determinar el conjunto C:

C = { z$\in$N : z divide a x $\wedge$ z es primo}

Dado que en este punto los términos en que está expresado el enunciado del problema están bien definidos, es decir, sabemos lo que es un número primo y sabemos cuando un número natural z divide a un número natural x, entonces no es necesario continuar refinando.

La especificación correspondiente del programa sería:

[var x: natural;
  var C: conjunto de naturales;
  { x=X ∧ X > 1 }
  cálculo de primos
  { C = { z∈N : z divide a x ∧ z es primo} }
].

En general, decir que A es igual al conjunto { z∈B : z cumple P(z)}, es equivalente a decir que A es subconjunto de B y (∀z: z ∈ B: (z ∈ A ≡ P(z))).

Por lo tanto, la especificación anterior también la podemos escribir como sigue:

[const X: natural;
  var C: conjunto de naturales;
  { X > 1 }
  cálculo de primos
  { (∀z: z es natural: (z ∈ C ≡ (z es primo)∧(z divide a X))) }
].

¿Qué diferencia hay entre el predicado (∀z: z es natural: (z ∈ C ≡ (z es primo)∧(z divide a X))) y el predicado (∀z: z es primo: (z ∈ C ≡ (z divide a X))) ?. Que en el segundo caso, C puede contener elementos que no son primos y el predicado ser verdad !!!. Por ejemplo para X=4, C={1,2,4}, el predicado ∀z: z es primo: ((z divide a X) ≡ (z ∈ C)) es verdadero, y C no es precisamente el que queremos calcule nuestro programa. El C que queremos es C={1,2}. Por lo tanto hay que tener cuidado en los rangos de las variables cuando usemos cuantificadores .

La especificación:

[const X: entero;
  var C: conjunto de enteros;
  { X > 1 }
  cálculo de primos
  { (∀z: z es primo: (z ∈ C ≡ (z divide a X))) }
].

es incorrecta pues sólo queremos que queden en C todos los números primos que dividen a X y ningún otro número natural.

2) Problema: Dada una secuencia de números enteros determinar el mayor número en la secuencia.

Se quiere colocar en una variable x el mayor número de una secuencia S dada.

Una primera especificación sería:

[const S: secuencia de enteros;
  var x: entero;
  { verdad }
  mayor valor
  {x está en la secuencia S y todo otro elemento de la secuencia tiene valor menor o igual que x}
]

Más formalmente, utilizando los operadores de secuencia:

[const S: secuencia de enteros;
  var x: entero;
  { verdad }
  mayor valor
  $\{ (\exists i: 0 \leq i < |S| : S[i]=x) \ \wedge (\forall i: 0 \leq i < |S| : S[i] \leq x) \}$
]

Note que la constante S hace referencia a un objeto de la clase (o tipo) secuencia.

Ejercicios:

1) Especifique un programa que calcule en una variable x el número de elementos distintos de cero de una secuencia s de N números enteros. (Utilice el cuantificador de conteo #, donde (#i: $0 \leq i < N$: P(i)) representa el número de enteros i entre 0 y N-1 que cumple P(i).
2) sección 1 guía de Michel Cunto y pag. 103 de Gries reemplazando la palabra arreglos por secuencias.

Reglas generales sobre especificaciones:

1) La precondición puede ser reforzada y la postcondición puede ser debilitada. Recordemos que si $P \Rightarrow Q$ es una tautología entonces decimos que P es más fuerte que Q ó que Q es más débil que P:

   Si {P} S {Q} se cumple y para todo estado $P_0 \Rightarrow P$ es verdad (es decir, $[P_0 \Rightarrow P]$), entonces $\{P_0\}$ S {Q} se cumple.
   Si {P} S {Q} se cumple y para todo estado $Q \Rightarrow Q_0$ es verdad (es decir, $[Q \Rightarrow Q_0]$), entonces {P} S $\{Q_0\}$ se cumple.

2) La expresión {P} S { falso } es equivalente a decir que para cualquier estado, P siempre es falso independientemente de S.

La primera regla nos dice que si tenemos un programa S que cumple {P} S {Q} y queremos encontrar un programa S' que cumpla $\{P_0\}$ S' {Q}, basta con tomar a S como S',

pues se cumple $\{P_0\}$ S $\{Q\}$. Por ejemplo, un programa S que cumple la especificación siguiente:

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

También cumplirá la especificación siguiente:

[var x,y : entero;
  { x=X ∧ y=Y ∧ X>10 }
  S
  { z=max(X,Y) }
].

Ya que (x=X ∧ y=Y ∧ X>10) ⇒ (x=X ∧ y=Y) es una tautología.

3) Regla de conjunción:

{P} S {Q}se cumple y {P} S {R} se cumple es equivalente a {P} S {Q∧R} se cumple

4) Regla de la disyuntividad:

{P} S {Q} se cumple y {R} S {Q} se cumple es equivalente a {P∨R} S {Q} se cumple

Note que Q∧R es más fuerte que Q y que R, y P∨R es más débil que P y que R.

3.2. Diseño de la solución algorítmica

Debemos encontrar un algoritmo que partiendo del estado requerido en la precondición, una vez que éste se ejecute, se obtenga el estado deseado en la postcondición. Las estrategias para llevar esto a cabo pueden ir desde utilizar sólo la intuición, el sentido común y estrategias generales de solución de problemas, hasta utilizar sólo "reglas precisas" para derivar (o calcular) el algoritmo a partir de la especificación (estas técnicas las veremos más adelante).

Entre las estrategias generales para resolver problemas se encuentra el *diseño descendente*. Este consiste en expresar la solución del problema como una composición de las soluciones de problemas más "sencillos" de resolver. El ejemplo de la sección 1.1. puede ser descompuesto en determinar cuántos litros de gasolina por kilómetro consume el vehículo y luego multiplicar esa cantidad por el número de kilómetros recorridos x. El problema fue descompuesto en dos subproblemas, el primero tiene solución l/k y el segundo (l/k)*x. El

diseño descendente también se denomina técnica de "refinamiento sucesivo" ya que, a su vez, a los problemas más sencillos se les aplica diseño descendente para resolverlos.

Una *máquina* es un mecanismo capaz de ejecutar un algoritmo expresado en función de las acciones elementales que ésta pueda ejecutar.

Cuando resolvemos un problema mediante la técnica de diseño descendente, vamos refinando la solución del problema en términos de algoritmos cuyas acciones elementales pueden ser ejecutadas por máquinas cada vez menos abstractas. Cuando un algoritmo puede ser ejecutado por una máquina real (el computador) decimos que el algoritmo es un programa. Un programa es un algoritmo destinado a comandar una máquina real.

Hallar un programa, para ser ejecutado por un computador, que resuelve un problema dado, significa aplicar la técnica de diseño descendente hasta encontrar un algoritmo, solución del problema, cuyas acciones pueden ser ejecutadas por el computador.

Por lo tanto el método consiste en definir máquinas abstractas cuyo funcionamiento es descrito por los algoritmos adaptados a cada nivel de máquina. Se parte de una máquina que trata información de alto nivel y es capaz de ejecutar acciones complejas. Habiendo descrito la solución del problema en términos de esas acciones, se toma cada una de éstas y se describen en función de acciones ejecutables por máquinas abstractas de nivel más bajo (es decir que manejan información y ejecutan acciones cada vez menos complejas). Cuando una máquina abstracta coincide con la real (por ejemplo, el lenguaje de programación que utilizamos), el análisis se termina. En este proceso de ir de máquinas abstractas a otras de menor nivel, se debe identificar analogías con problemas ya resueltos y utilizar soluciones ya elaboradas para no tener que reinventar la rueda!

En el ejemplo de la sección 1.1. una primera solución del problema consistió en descomponer el problema original en dos subproblemas a resolver de manera secuencial: primero calcule el consumo de gasolina por kilómetro, luego con el consumo por kilómetro y la cantidad de kilómetros a recorrer x calcule el gasto de gasolina. Por lo tanto si contamos con una máquina que sepa resolver estos dos subproblemas, ya conseguimos la solución. En caso contrario, tendremos que descomponer cada uno de los dos subproblemas en problemas "más sencillos", por ejemplo determinar el consumo por kilómetro se puede expresar como dividir la cantidad 1 entre k, ¿pero sabemos dividir?.... Utilizaremos y ejercitaremos la técnica de diseño descendente en los ejemplos que iremos presentado en las notas.

<u>Ejemplo de aplicación de diseño descendente:</u>

Problema: Estoy en casa, deseo leer el periódico y no lo tengo.

Una solución sería:

(1) Salir de casa.
(2) Ir al puesto de venta del periódico y comprarlo.
(3) Regresar a casa.
(4) Leer el periódico.

Podemos *refinar* aún más las acciones dependiendo de si el ejecutante del algoritmo sabe o no realizar cada una de las acciones descritas. Por ejemplo, podemos precisar mejor la acción "Salir de la casa":

Salir de la casa:

(1) Verificar que se tiene dinero suficiente para comprar el periódico
(2) Ir hasta la puerta principal de la casa.
(3) Abrir la puerta.
(4) Salir al exterior de la casa.
(5) Cerrar la puerta.

Un computador es capaz de ejecutar acciones muy elementales: operaciones aritméticas y lógicas, y controlar el flujo de ejecución. Por esta razón existen los lenguajes de programación, para liberar al programador de la ardua tarea de escribir los programas en el lenguaje que reconoce el computador (el lenguaje de máquina), y así poder escribir los programas en un lenguaje de más alto nivel de abstracción, más fácil de utilizar por el programador. Un *lenguaje de programación*, como JAVA, PASCAL, C, no es más que un repertorio de acciones elementales de una máquina abstracta, estas acciones tienen un nivel de abstracción más alto que las acciones que puede ejecutar el computador.

Para que un computador pueda ejecutar las acciones de un programa escrito en un lenguaje de alto nivel, es necesario que exista un traductor que traduzca el programa de alto nivel a un programa (o código) que pueda comandar al computador. A este traductor se le conoce con el nombre de *Compilador*. Otra forma en que un computador puede ejecutar un programa escrito en un lenguaje de alto nivel es a través de otro programa, llamado *Interpretador*, que simula la máquina virtual que representa el lenguaje de alto nivel. El interpretador analiza y luego ejecuta las acciones del programa escrito en el lenguaje de alto nivel.

Un algoritmo escrito en un lenguaje de programación de alto nivel, lo llamaremos programa, pues existe otro mecanismo automatizado (el compilador) que permite traducirlo a lenguaje de máquina. Un lenguaje de programación constituye un medio para hacer programas de una manera precisa y rigurosa, pues cada acción elemental del lenguaje tiene reglas sintácticas (cómo se escriben las acciones) y reglas semánticas (qué significa desde el punto de vista operacional la acción) tan precisas como el lenguaje matemático. Hacer un programa correcto exige la misma rigurosidad que hacer una demostración matemática de un teorema.

Por lo tanto es importante distinguir claramente dos etapas en la actividad de programación: una, desarrollar el algoritmo; la otra, refinar el algoritmo hasta convertirlo en un programa.

Ejercicio: Describir en lenguaje natural el proceso aprendido en la escuela, para dividir un número X entre otro Y. Las acciones elementales deberán ser suficientemente claras para que otra persona (la máquina que lo ejecutará) las pueda llevar a cabo.

En las notas utilizaremos un lenguaje para describir nuestros programas, que además de tener acciones elementales precisas, también permite tener acciones descritas en lenguaje natural, por lo que lo llamaremos *pseudolenguaje*. Los datos iniciales serán proporcionados por una instrucción de lectura, y los resultados se escribirán mediante una instrucción de escritura, de la forma leer(...) y escribir(...). En general, no haremos mención explícita de estas instrucciones pues enfocaremos nuestra atención en el diseño del algoritmo que describe el cómputo de los resultados en términos de la entrada sin importar de donde provienen los datos de entrada ni para qué serán utilizados los resultados.

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.

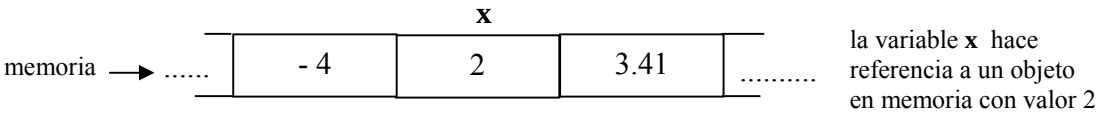


Figura 4

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.

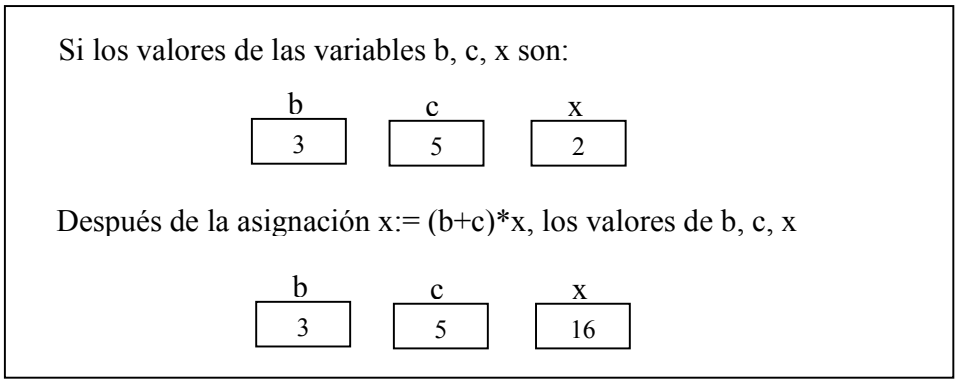


Figura 5

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⇒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 la siguiente tripleta de Hoare:

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

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:

$(x \geq 0)(x:=x+1)$
$\equiv$ definición de la regla de sustitución (sustituir todas las apariciones de x por x+1)
$x+1 \geq 0$
$\equiv$ por aritmética
$x \geq -1$
$\Leftarrow$ 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:

3) 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 ∧ y= X+Y

Con esto último hemos probado que:

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

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

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

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

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

Lo anterior nos dice que se cumple lo siguiente:

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

Ejercicios:

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:

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

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;
{ $N \ge 0$ }
S
{($N = 86.400*d + 3.600*h + 60*m + s$) $\wedge$
$0 \le s < 60 \wedge 0 \le m < 60 \wedge 0 \le h < 24 \wedge 0 \le d$ }
].

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
$N = 86.400 * (N \text{ div } 86.400) + N \bmod 86.400 \quad \wedge \quad 0 \leq N \bmod 86.400 < 86.400 \ \wedge \ 0 \leq N \text{ div } 86.400$
⇐ 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:

$$0 \le \mathbf{rd} \text{ div } 3.600 < 86.400 \text{ div } 3.600 = 24$$

Por otro lado, por definición de div y mod:

$$\mathbf{rd} = 3.600 * \mathbf{rd} \text{ div } 3.600 + \mathbf{rd} \text{ mod } 3.600$$

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

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

$(Y = 86.400*dy + 3.600*hy + 60*my + sy) \land 0 \leq sy < 60 \land 0 \leq my < 60 \land 0 \leq hy < 24$
$\land 0 \leq dy \land$
$(Z = 86.400*dz + 3.600*hz + 60*mz + sz) \land 0 \leq sz < 60 \land 0 \leq mz < 60 \land 0 \leq hz < 24$
$\land 0 \leq dz \quad \}$
].

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 ≥ 0, Y ≥ 0, Z ≥ 0 }
         S1;
```

S2;
S3
{ (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 }

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∧R} S {Q∧R} se cumple.

En nuestro caso tenemos que originalmente se cumple lo siguiente:

- { X ≥ 0 } S1 { R1: (X = 86.400*dx + 3.600*hx + 60*mx + sx) ∧ 0 ≤ sx < 60
∧ 0≤mx< 60 ∧ 0 ≤hx <24 ∧ 0 ≤dx }

- { Y ≥ 0 } S2 { R2: (Y = 86.400*dy + 3.600*hy + 60*my + sy) ∧ 0 ≤ sy < 60
∧ 0≤my< 60 ∧ 0 ≤hy <24 ∧ 0 ≤dy }

- { Z ≥ 0 } S3 { R3: (Z = 86.400*dz + 3.600*hz + 60*mz + sz) ∧ 0 ≤ sz < 60
∧ 0≤mz< 60 ∧ 0 ≤hz <24 ∧ 0 ≤dz }

Tomando a R, en la regla de fortalecimiento, como Y ≥ 0 ∧ Z ≥ 0, se cumple entonces:

{ Q1: X ≥ 0 ∧ Y ≥ 0 ∧ Z ≥ 0}

S1

{ Q2: R1 ∧ Y ≥ 0 ∧ Z ≥ 0}

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

Z ≥ 0 ∧ R1

se cumple:

{ Q2: Y ≥ 0 ∧ Z ≥ 0 ∧ 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:

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

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

entrada <lista de identificadores separados por coma> : <tipo>
entrada-salida <lista de identificadores separados por coma> : <tipo>
salida <lista de identificadores separados por coma> : <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 después 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 los parámetros reales de 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 parámetros reales (o 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 después 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:

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

Lo anterior lo denominaremos *encabezado* del procedimiento. Las pre y post condiciones indican "qué hace" el procedimiento y el cuerpo es "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", es decir, variables cuya declaración (o definición) no aparece en la especificación de los parámetros, ni en el cuerpo del procedimiento). 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 $\wedge$ N ≥ 0 }
{ **Post**: (N = 86.400*d + 3.600*h + 60*m + s) $\wedge$ 0 ≤ s < 60 $\wedge$ 0≤m< 60 $\wedge$ 0 ≤h <24
$\wedge$ 0 ≤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.

4.3.2. 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 formales 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 debe cumplir que los identificadores de entrada-salida y de salida deben ser distintos dos a dos. Tercero, 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).

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 respectivamente los valores resultantes a las variables $x_1, x_2, \ldots, x_n$. Exigimos además que las variables $x_1, x_2, \ldots, x_n$ sean distintas dos a dos (esto evita el conflicto de cual será el valor final de la variable después de ejecutada la asignación múltiple, si esa variable aparece varias veces al lado izquierdo de la asignación). Lo anterior garantiza 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:

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

Queda entendido entonces, de acuerdo a la definición de asignación múltiple, que los identificadores de las variables en b, c deben ser distintos entre si, es decir, no puede haber dos parámetros reales (correspondientes a parámetros de salida o entrada-salida) con el mismo identificador.

4.3.3. Reglas de llamada a procedimiento

Dado el procedimiento:

proc p(**entrada** x; **entrada-salida** y; **salida** z)
{ P }
{ Q }
[S]

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

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

donde p(a, b, c) es una llamada al procedimiento p.

4.3.3.1. Caso A: Procedimientos con parámetros sólo de entrada y de salida

Consideremos primero el caso más sencillo donde todos los parámetros son sólo de entrada y/o de salida, es decir, no hay parámetros de entrada-salida.

Supongamos que se tiene la siguiente definición del encabezado de un procedimiento:

proc p(**entrada** x; **salida** y)
{ Pre: Pdef }
{ Post: Qdef }

donde:

- x es un parámetro de entrada, y es un parámetro de salida.
- x es considerado constante.
- En Pdef sólo x puede ocurrir libre.
- En Qdef sólo x e y pueden ocurrir libres.
- No se permite efectos de borde: el cuerpo sólo manipula a x, a y, y a variables locales.

Notemos lo siguiente:

(i) Hemos omitido los tipos de los parámetros formales x, y. Tal información de tipo es sólo relevante para determinar parámetros reales adecuados, ya que sólo se permite

llamadas en las que el tipo de los parámetros reales coincede con el tipo de los parámetros formales correspondientes. Aparte de esta verificación de adecuación de parámetros formales con reales, los tipos son irrelevantes para la definición de las reglas de correctitud; por esto los omitimos.

(ii) Consideramos sólo un parámetro de entrada y sólo uno de salida para lograr enunciar las reglas de correctitud de la manera más simple posible. Se puede generalizar fácilmente las reglas cuando hay mayor cantidad de parámetros. Es posible considerar tanto a $\underline{x}$ como a $\underline{y}$, como listas de parámetros $\langle x_i \rangle$ y $\langle y_i \rangle$, asumiendo así que $\underline{x}$ e $\underline{y}$ contemplan el caso general.

(iii) Si el procedimiento es implementado con el cuerpo S, éste debe satisfacer la tripleta {Pdef} S {Qdef}. De allí que podamos, y nos convenga, ignorar el cuerpo, pues la única información relevante sobre él es que es correcto respecto a la precondición Pdef y la postcondición Qdef.

A continuación daremos las reglas de correctitud de llamadas para el caso que nos ocupa. Presentaremos dos versiones: la primera versión es sencilla pero lamentablemente incompleta (desde el punto de vista formal de "completitud"), la segunda versión es más compleja pero completa.

Sea p(E,a) una llamada al procedimiento, donde E es una expresión y a es una variable. Note que los parámetros reales de entrada siempre pueden ser expresiones cualesquiera, mientras los parámetros reales de salida deben denotar variables. Note también, que la variable a podría ocurrir (libre) en la expresión E. Esto genera complicaciones en las reglas, por lo cual consideramos primero el caso en que no ocurre tal solapamiento entre la entrada y la salida de la llamada.

Regla A-1:

Con respecto a la definición dada del procedimiento p, la tripleta de Hoare:

$$\{Pllam\}\ p(E,a)\ \{Qllam\}$$

donde a no ocurre (libre) en E, se cumple **si** se cumplen las siguientes condiciones:

1) $Pllam \Rightarrow Pdef(x:=E)$ es una tautología.
2) $Qdef(x,y := E,a) \Rightarrow Qllam$ es una tautología

Ejemplo:

Si tenemos el procedimiento con encabezao:

proc duplicar(**entrada** e:entero; **salida** s:entero)
{Pre: $e \geq 0$}
{Post: $s = 2*e$ }

usando la regla A-1 se puede demostrar la correctitud de las siguientes llamadas:

- { n ≥ 0 } duplicar(n,m) { m = 2*n }
- {true} duplicar(n*n,m) {m= 2*n*n }
- { n ≥ -5 } duplicar(n+10,m) {m mod 2 = 0 }

Se deja como ejercicio verificar que las dos condiciones de la regla A-1 se cumplen en los tres casos.

Ahora bien, esta primera regla es incompleta, en el sentido de que ciertas tripletas de Hoare no pueden ser demostradas por ella. Hay dos problemas con esta regla:

– Primero, como la segunda condición sólo relaciona a Qdef con Qllam, no se puede utilizar en el estado final información del estado inicial (justo antes de la llamada a P) que aún sea válida. Por ejemplo, las siguientes tripletas son correctas y no pueden ser demostradas por nuestra regla:

  { n ≥ 5 } duplicar(n,m) { m ≥ 10 }
  { a > 7 } duplicar(a + b*b, c) { c ≥ 16 }
  { 10 < m < n } duplicar(n,m) { m ≥ 24 }
  { n ≥ 5 y m ≥ 5 } duplicar(n,m) { m > 0 }

  Se deja como ejercicio al lector verificar que las condiciones de la regla A-1 no son satisfechas. La que siempre falla es la segunda, que es la que adolece de no acarrear información del estado inicial hacia el final.

– El segundo problema se refiere a la restricción impuesta sobre ocurrencias de **a** en E, lo cual deja de considerar llamadas en la que se solape la entrada con la salida. Ejemplos de llamadas correctas no consideradas incluyen las siguientes:

  { n ≥ 5 } duplicar(n,n) { n ≥ 10 }
  { a > 7 } duplicar(a + b*b, a) { a ≥ 16 }
  { a > 7 } duplicar(a + b*b, b) { b ≥ 16 }
  { 0 < k ≤ N ∧ N-k = X } duplicar(k, k) { N-k < X }

  En estos casos, ni siquiera tiene sentido intentar verificar las condiciones de la Regla A-1, pues el enunciado de la regla indica que no se debe aplicar en estos casos.

Veamos entonces la nueva regla. Para resolver el primr problema referente a acarrear información del estado inicial, que aún es válida en el estado final, usaremos a Pllam en la segunda condición. Como la variable de salida **a** pierde su información inicialcambiaremos **a** en Pllam por un nuevo nombre no utilizado. Por conveniencia, usaremos la variable de especificación A (la cual la podremos imaginar como “el valor inicial de **a** “). Para resolver el segundo problema de solapamiento de la entrada con la salida, en la sustitución sobre Qdef tambien cam biaremos el estado inicial de **a** por A. A continuación la nueva regla:

Regla A-2:

Con respecto a la definición dada del procedimiento P, la tripleta de Hoare

$$\{ \text{Pllam} \}\ \ P(E,\mathbf{a})\ \ \{ \text{Qllam} \}$$

se cumple sí y sólo si se cumplen las siguientes condiciones:

$$\text{Pllam} \Rightarrow \text{Pdef}(x:=E)\ \text{ y}$$
$$\text{Pllam}(\mathbf{a}:=A) \wedge \text{Qdef}(x,y := E(\mathbf{a}:=A),\mathbf{a}) \Rightarrow \text{Qllam}$$

donde A es una nueva variable de especificación.

Ejemplos:

Los 8 ejemplos de llamadas ya dados como no atrapados por la primera regla A-1, sí son bien manejados por la segunda regla A-2 (los 8 ejemplos son los 4 del primer problema, y los otros 4 del segundo problema). Se deja como ejercicio al lector verificar que las dos condiciones de la regla A-2 efectivamente se cumplen en los 8 casos.

Note que la regla A-1 estaba enunciada como un condicional sí, la cual la hacía una regla incompleta, mientras que la regla A-2 con su sí y sólo si es una regla completa para el caso A (procedimientos con parámetros sólo de entrada y salida).

Ejercicios:

1) Suponga que cuenta con procedimiento de exponenciación:

proc exponenciar(entrada base, exp: int; salida result:int)
{ Pre: exp ≥ 0 }
{ Post: result = base ^{exp} }

y se tiene un problema computacional especificado como sigue:

[const dato0, dato1, dato2, dato3, dato4, dato5: int;
var resA, resB, resC : int;
{ dato1 ≥ 0 ∧ dato3 ≥ 0 ∧ dato5 ≥ 0 }
TresExp
{ resA = dato0^{dato1} ∧ resB = dato2^{dato3} ∧ resC = dato4^{dato5} }
]

Demuestre que el problema TresExp se puede resolver con la instrucción:

exponenciar(dato0, dato1, resA);
exponenciar(dato2, dato3, resB);
exponenciar(dato4, dato5, resC)

Recuerde que para demostrar cada secuenciación (;) hacen falta aserciones intermedias, que a la vez serán usadas en la demostración de correctitud de cada llamada. Estas deben ser propuestas por intuición, salvo que sean calculadas por wp (precondición más débil). Use como primera aserción intermedia a:

$$\{ resA = dato0^{dato1} \wedge dato3 \geq 0 \wedge \ dato5 \geq 0 \}$$

sugiera usted su propia segunda aserción intermedia y complete la demostración de correctitud.

2) Usando el mismo procedimiento exponenciar del ejercicio anterior, se quiere resolver el siguiente problema:

```
[ const a, b, c : int;
  var r : int;
   { b ≥ 0 ∧  c ≥ 0 }
    ExpSobreExp
   { r =   a^(b^c)}
]
```

Donde **a^(b^c)** es el resultado de elevar **a** a la potencia **b^c** (**b** a la potencia **c**).

Demuestre la correctitud de las siguientes dos soluciones al problema ExpSobreExp:

```
[   var temp : int;
    exponenciar(b,c,temp);
    exponenciar(a,temp,r);
]
y
    exponenciar(b,c,r);
    exponenciar(a,r,r);
```

3) Con la misma definición de encabezado para exponenciar, demuestre la correctitud del siguiente programa:

```
[  const N : int;
         a : array [0..N) of int;
  var r : int;
  { N > 0 }
  [ var k:int;
    r, k := a[0], 1;
    { Invariante: 1 ≤ k ≤ N ∧ r = (max i: 0 ≤  i < k : a[i]^(i+1) )}
    do k != N →
      [  var temp : int;
         exponenciar(a[k],k+1,temp);
         r := r max temp;
```

```
        k := k + 1;
      ]
    od
  ]
  { r = (max i: 0 ≤ i < N : a[i]^(i+1) ) }
]
```

4) Construya otra solución iterativa para el mismo problema de la pregunta anterior, pero usando el invariante correspondiente a una iteración "descendente", esto es, el invariante:

$0 \le k \le N-1 \wedge r = (\max i: k \le i < N : a[i]^{(i+1)})$

Demuestre la correctitud de su solución.

5) La regla A-2 resuelve dos problemas de la regla A-1 tal como fue explicado en el texto: usar información proveniente del estado inicial en el estado final, y permitir solapamiento entre los parámetros reales de entrada y los parámetros reales de salida.

El primer problema ha podido tratarse por separado, aunque en el texto se resolvió resolverlo en conjunto con el segundo. Enuncie usted una regla A-1' en la que se resuelve sólo el primer problema pero no el segundo: esto es, aún se prohibirá ocurrencias del parámetro real de salida en el parámetro real de entrada, pero Pllam será combinado con Qdef.

6) Muestre que con su regla A-1', se puede demostrar la correctitud de:

- Las cuatro llamadas:
  - { n ≥ 5 } duplicar(n,m) { m ≥ 10 }
  - { a > 7 } duplicar(a + b*b, c) { c ≥ 16 }
  - { 10 < m < n } duplicar(n,m) { m ≥ 24 }
  - { n ≥ 5 y m ≥ 5 } duplicar(n,m) { m > 0 }
- El programa del ejercicio 1
- La primera solución del ejercicio 2, pero no la segunda.
- El programa del ejercicio 3.

4.3.3.2 Caso B: Procedimientos con todo tipo de parámetros: entrada, entrada-salida, salida

Supongamos que se tiene la siguiente definicion (del encabezado) de un procedimiento:

```
proc P(entrada x; entrada-salida y; salida z)
    { Pre:  Pdef }
    { Post: Qdef }
```

Donde:

- **x** es considerado constante

- En Pdef solo puede ocurrir libre **x, y**
- En Qdef sólo pueden ocurrir libres **x**, **y_0, y, z**
- No se permiten efectos de borde, es decir, el cuerpo sólo manipula **x**, **y, z** y sus variables locales
- Note de nuevo, como en el caso A de la sección 4.3.3.1, que:
  i) Hemos omitido los tipos de los parámetros formales **x**, **y, z**, por las mismas razones que antes.
  ii) Las reglas son fácilmente generalizables a mayor cantidad de parámetros. También es posible considerar tanto a **x, y , z** como vectores de parámetros $\mathbf{x}_0$, $\mathbf{x}_1$, …, $\mathbf{y}_0$, $\mathbf{y}_1$, …., $\mathbf{z}_0$, $\mathbf{z}_1$, …., asumiendo así que ya **x**,**y**,**z** contemplan el caso general.
  iii) Si el procedimiento es implementado con cuerpo I, este debe satisfacer la tripleta {Pdef} I {Qdef}.

Al igual que en el caso A de la sección 4.3.3.1, presentaremos primero una regla incompleta, y luego otra completa pero más compleja. Las llamadas serán de la forma P(E,**a**,**b**) con E una expresión cualquiera (aunque del tipo adecuado) y **a**, **b** variables. La primera regla o contemplará solapamiento entre la entrada y la salida.

Regla B-1

Con respecto a la definición dada del procedimiento P, la tripleta de Hoare:

$$\{ \text{Pllam} \} \quad P(E,\mathbf{a},\mathbf{b}) \quad \{ \text{Qllam} \}$$

donde **a, b** no ocurren (libres) en E, se cumple si se cumplen las siguientes dos condiciones:

$$\text{Pllam} \Rightarrow \text{Pdef}(\mathbf{x},\mathbf{y}:=E,\mathbf{a}) \text{ y}$$
$$\text{Qdef}(\mathbf{x}, \mathbf{y_0}, \mathbf{y}, \mathbf{z} := E,A,\mathbf{a},\mathbf{b}) \Rightarrow \text{Qllam}$$

donde A es una nueva variable de especificación.

Note el manejo del nuevo tipo de parámetro entrada-salida con respecto a la regla A-1. En la primera condición, **y** es tratado igual al parámetro de entrada **x**, mientras que en la segunda condición, **y** es tratado como el parámetro de salida **z**. Sobre el manejo de $\mathbf{y_0}$, note que este no puede ser asociado a **a**, pues lo deseable es que en el estado final, **a** tenga un nuevo valor y haya perdido su valor inicial; como se hizo en la regla A-2, al valor inicial de a le asociamos una nueva variable de especificación A (asumiendo que este identificador de variable no está en uso al momento).

Ejemplo:

Con el procedimiento:

proc duplicar(entrada-salida num : int)
{ Pre: num ≥ 0 }
{ Post: num = 2*num_0 }

Podemos, mediante la regla B-1, probar la tripleta:

$$\{ n > 7 \}\ \text{duplicar}(n)\ \{ n \bmod 2 = 0 \}$$

se deja como ejercicio al lector verificar este hecho.

Ahora bien, tal como en el caso A, necesitamos una nueva regla B-2, que sea completa, permitiendo que se acarree información inicial hacia el estado final y permitiendo tambien que parámetros reales de entrada (in) usen parámetros reales de entrada-salida o de salida (entrada-salida ó salida respectivamente). Utilizaremos el mismo truco de valernos de nuevas variables de especificación A, B para referirnos a los valores iniciales (perdidos en el estado final) de los parámetros reales de salida **a, b**.

Veamos la nueva regla,

Regla B-2:

Con respecto a la definición dada del procedimiento P, la tripleta de Hoare:

$$\{ \text{Pllam} \}\ \ P(E,\mathbf{a,b})\ \ \{ \text{Qllam} \}$$

se cumple si y sólo si se cumplen las siguientes dos condiciones:

$$\text{Pllam} \Rightarrow \text{Pdef}(\mathbf{x},\mathbf{y}{:=}E,\mathbf{a})\ \text{y}$$
$$\text{Pllam}(\mathbf{a,b}{:=}A,B) \wedge \text{Qdef}(\mathbf{x, y_0, y, z :=}\ E(\mathbf{a,b}{:=}A,B)\mathbf{,A,a,b}) \Rightarrow \text{Qllam}$$

donde A y B son nuevas variables de especificación.

Ejemplos:

Con el procedimiento duplicar del ejemplo anterior, la nueva regla B-2 nos permite probar las siguientes tripletas:

- $\{ n \geq 5 \}\ \text{duplicar}(n)\ \{ n \geq 10 \}$
- $\{ 0 < k \leq N \wedge\ N-k = X \}\ \text{duplicar}(k)\ \{ N-k < X \}$
- $\{ p \geq 0 \wedge\ p \bmod 2 = 0 \}\ \text{duplicar}(p)\ \{ p \bmod 4 = 0 \}$
- $\{ p \geq 0 \wedge\ p \bmod 2 = 1 \}\ \text{duplicar}(p)\ \{ p \bmod 4 = 2 \}$

Se deja como ejercicio al lector verificar que las dos condiciones indicadas en la regla B-2 se cumplen en estos cuatro casos.

Ejercicios:

1) Suponga que cuenta con un nuevo procedimiento de exponenciación, definido como sigue:

proc exponenciar(entrada exp: int; entrada-salida num:int)

{ Pre: $exp \geq 0$ }

{ Post: $num = num_0^{exp}$ }

Con este nuevo procedimiento, el problema dado por la especificación:

[const dato0, dato1, dato2, dato3, dato4, dato5: int;
  var resA, resB, resC : int;
  { $dato1 \geq 0 \wedge dato3 \geq 0 \wedge dato5 \geq 0$ }
  TresExp
  { $resA = dato0^{dato1} \wedge resB = dato2^{dato3} \wedge resC = dato4^{dato5}$ }
]

Se puede resolver con la instrucción:

resA := dato0; exponenciar(dato1, resA);
resB := dato2; exponenciar(dato3, resB);
resC := dato4; exponenciar(dato5, resC)

Demuestre la correctitud de esta solución. Recuerde que para demostrar cada secuenciación (;) hacen falta aserciones intermedias, que a la vez serán usadas en la demostración de correctitud de cada llamada. Estas deben ser propuestas por intuición, salvo que sean calculadas por wp (precondición más débil). Sugerimos que use como primera aserción intermedia a:

{ $resA = dato0 \wedge dato0 \geq 0 \wedge dato3 \geq 0 \wedge dato5 \geq 0$ }

y como la segunda a:

{ $resA = dato0^{dato1} \wedge dato3 \geq 0 \wedge dato5 \geq 0$ }

Proponga usted las aserciones intermedias restantes y complete la demostración de correctitud.

2) Demuestre que con el nuevo procedimiento exponenciar definido en el ejercicio anterior, el problema ExpSobreExp del ejercicio 2 de la sección anterior se puede resolver correctamente con la instrucción siguiente:

```
[  var temp : int;
   r, temp := a, b;
   exponenciar(c,temp);
   exponenciar(temp,r);
]
```

3) Complete adecuadamente en el siguiente programa a la postcondición y el invariante:

```
[  const a, b, c : int;
   var r : int;
   { b ≥ 0 ∧ c ≥ 1 }
   r := a;
   exponenciar(b, r);
   [
     var k : int;
     k := 1;
     { Invariante: 1 ≤ k ≤ c ∧ r = ???
       función de cota : c-k}
     do k != c →
          exponenciar(b,r);
          k := k + 1;
     od
   ]
   { r = ????  }
]
```

4) Muestre que el problema resuelto por el programa anterior ha podido ser resuelto por un programa compuesto sólo por una asignación y una llamada.

4.3.4. Otros ejemplos de demostración de la correctitud de llamadas a procedimientos.

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

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

Queremos probar que se cumple:

(1) { a = X ∧ b = Y } intercambio (a, b) { a = Y ∧ 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 ∧ b = Y) ⇒ (y1 = X ∧ y2 = Y) (y1, y2 := a,b)
- (A1 = X ∧ A2 = Y) ∧ (a = A2 ∧ b = A1) ⇒ (a = Y ∧ b = X)

lo cual es evidente.

Ejemplo2:

Queremos probar lo siguiente:

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

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

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

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

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

1) $(x = X \wedge X \geq 0) \Rightarrow (n \geq 0) (n:=x)$

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

lo cual resulta evidente.

Las variables de especificación que denotan valores iniciales y finales de los parámetros, 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. Por ejemplo, si la precondición del procedimiento anterior hubiese sido $n=N \wedge N \geq 0$, aunque utilizar una variable de especificación para denotar el valor inicial de un parámetro formal de entrada sea innecesario, y la postcondición hubiese sido: $(N = 86.400*d + 3.600*h + 60*m + s) \wedge 0 \leq s < 60 \wedge 0 \leq m < 60 \wedge 0 \leq h < 24 \wedge 0 \leq d$, entonces hemos podido cambiar N por X para adecuarla a la demostración de correctitud.

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: verdad }
        {Post: z = x^2 + 3x – 2  }
        [
          z := x*x + 3x – 2
        ]
```

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.

Ejercicio: 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.

Solución parcial:

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".

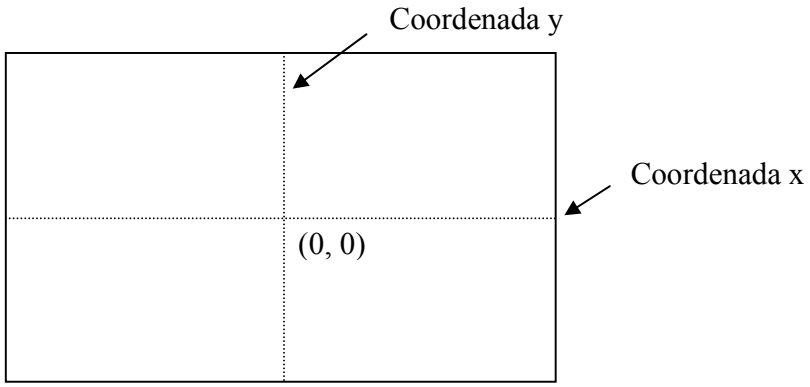


Figura 6

Los métodos son los siguientes:

proc DibujarSegmento(entrada x1, y1, x2, y2 : real)
{ 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) }

proc DibujarCírculo(entrada x, y, r : 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 }

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 ≤ 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) ∧ 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| ≤ XMAX  ∧ |y| ≤ YMAX  ∧  0 ≤ long  ∧  0 ≤ |x| + long/2 ≤ XMAX
                 ∧  0 ≤ |y| + long/2 ≤ 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.

4.3.5. Funciones

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:

```
func  f ( x : T)  →  T'
        { Pre: Pdef }
        { Post: Qdef }
        [
          I
            >> E
        ]
```

Donde:

- **x** es considerado constante
- En Pdef solo puede ocurrir libre **x**
- En Qdef sólo pueden ocurrir libres **x**, **f**; representando el identificador **f** a un pseudo-parámetro de salida correspondiente al valor devuelto por la función.
- No se permiten efectos de borde, es decir, el cuerpo sólo manipula **x**, y sus variables locales
- Note que:
  - i) Esta vez hemos incluido la información de tipos, pues eso es lo único en el encabezado que representa al valor de salida.
  - ii) Consideramos sólo un parámetro de entrada. Las reglas son luego fácilmente generalizables a mayor cantidad de parámetros. También es posible considerar a **x** como un vector de parámetros $\mathbf{x}_0$, $\mathbf{x}_1$, …, cada uno con su propio tipo, asumiendo así que ya **x** contempla el caso general.
  - iii) La correctitud del cuerpo corresponde a que en el estado final de I, la expresión E dé un valor adecuado para el resultado de **f** según su postcondición. Esto es, lo que Qdef exige a **f** debe ser satisfecho por E en el estado final de I. En términos de tripletas de Hoare esto significa { Pdef } I { Qdef(f := E) }.

Un ejemplo de declaración de función sería:

```
func F( x: real) → real
    {Pre: true }
    {Post:  F = x^2 + 3x – 2  }
    [
      var z : real;
```

```
    z := x^2 + 3*x -2

    >> z
  ]
```

Cuando la secuencia de acciones I es vacía, podemos colocar:

```
func F(x: real) → real
    {Pre: true }
    {Post:  F = x^2 + 3x – 2  }
     = x^2 + 3*x -2
```

Indicando de esta forma que el valor de la función es la expresión x^2 + 3*x -2

4.3.5.1. Definición formal de una llamada a función

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

```
func  f ( x : T)  →  T'
        { Pre: Pdef }
        { Post: Qdef }
        [
          I
            >> E
        ]
```

donde $\underline{x}$ puede representar una lista de los parámetros de entrada x_i , con sus respectivos tipos, separados por ; .

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*(2-s)). Los a_i son los argumentos de entrada, o parámetros reales, que se corresponden respectivamente a los parámetros formales x_i. Cada argumento debe ser del mismo tipo que su parámetro formal correspondiente.

Una llamada a la función F del ejemplo dado en la sección anterios 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 formales 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. En el estado final del cuerpo I, la expresión E representa un valor adecuado para el resultado de **f** según su postcondición Qdef. Esto es, lo que Qdef exige a **f** debe ser satisfecho por E en el estado final de I. En términos de tripletas de Hoare esto significa que la tripleta { Pdef } I { Qdef(f := E) } es correcta. El valor que representa F($\underline{a}$) en el programa que llama a la función es el que resulta de evaluar esta expresión E, y decimos que este valor es el que "devuelve" F aplicado a $\underline{a}$.

Una *llamada a función* es una expresión y no una instrucción, por lo que deberá formar parte de una expresión en una instrucción de nuestro programa, por ejemplo:

```
z := F(3+5*s)
        ó
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.

4.3.5.2 Reglas de llamada a función

Supongamos que tenemos la siguiente definición de función:

```
func  f ( x : T)  →  T'
        { Pre: Pdef }
        { Post: Qdef }
        [
          I
            >> E
        ]
```

Ahora bien, como una función es casi como un procedimiento con un solo parámetro de salida, definamos en general "el procedimiento asociado a una función", como sigue. Para el patrón general de función **f** antes dado, su procedimiento es:

```
proc Pf ( entrada x : T ; salida @ : T')
        { Pre: Pdef }
        { Post: Qdef (f := @ )}
        [
          I; @ := E
        ]
```

Donde @ es un identificador nuevo.

Note que la correctitud del cuerpo de este procedimiento **Pf** equivale a la correctitud de la función **f** antes indicada.

En relación con las llamadas a funciones, en vista de que tales llamadas no son instrucciones, sino expresiones, su correctitud no puede ser expresada directamente con tripletas de Hoare. La correctitud de tales llamadas debe manejarse indirectamente como la corretitud de la instrucción dentro de la cual aparezca la expresión de la llamada a función. Por ejemplo, una llamada a **f**(a), sabiendo que es una expresión, podría aparecer en una instrucción como:

(1) b:= 3*(f(a)-b)

o como:

(2)
```
if f(a) > 5 → ...
| f(a) ≤ 5 → ...
fi
```

o como:

(3) do (f(a) mod 2 = 0) → ... od

En todos los casos hay que re-expresar la instrucción haciendo uso del procedimiento **Pf** asociado a **f**. Esto es, previo al punto en que se usa a **f**, se realiza una llamada a **Pf**, dejando el resultado en una nueva variable ⊗. Luego se utiliza la variable ⊗ en el lugar en el que se requería el resultado de la función.

Tal cambio de **f** por **Pf** es más directo cuando la llamada a función no está en la guardia de una iteración. Veamos primero los dos ejemplos sencillos y luego el caso de guardia de iteración. Retomemos los tres ejemplos anteriores

Ejemplo 1: obtendríamos **Pf**(a,⊗) ; b:= 3*(⊗-b)

Ejemplo 2: obtendríamos **Pf**(a,⊗) ; if ⊗ > 5 → ... | ⊗ ≤ 5 → ... fi

Ejemplo 3: El caso un poco más complejo de guardia de iteración requiere cuidar que la primera llamada se hace antes de entrar a la iteración, mientras que las restantes llamadas se hacen al finalizar el cuerpo de la iteración. Esto es,

Pf(a,⊗) ; do ⊗ > 10 → ... ; **Pf**(a,⊗) od

Note que una misma instrucción podría tener varias llamadas a función, en cuyo caso se pueden resolver progresivamente. Por ejemplo:

c := 3*(**f**(a) – **f**(b))

se puede resolver una llamada mediante la receta anterior, obteniéndose:

$$\mathbf{Pf}(a,\otimes)\ ;\ c := 3*(\otimes - \mathbf{f}(b))$$

y ahora, en la instrucción a la derecha de la secuenciación volvemos a aplicar la receta para obtener:

$$\mathbf{Pf}(a,\otimes)\ ;\ \mathbf{Pf}(b,\circledR)\ ;\ c := 3*(\otimes - \circledR)$$

Es importante resaltar que de haber resuelto las llamadas en orden inverso, l programa resultante habría sido equivalente (pues las dosllamadas a **Pf** no interfieren la una con la otra).

Más interesante aún son ejemplos con anidamiento, como por ejemplo:

$$b := 3*\mathbf{f}(a - 2*\mathbf{f}(b))$$

Resolvamos detalladamente esta instrucción de dos maneras: primero, empezando por la llamada "interna" f(b) pasando luego a la otra; y, segundo, empezando por la llamada externa f(a ...):

Empezado por la llamada interna, resolvemos hacia:

$$\mathbf{Pf}(b,\otimes)\ ;\ b := 3*\mathbf{f}(a - 2*\otimes)$$

y luego se resuelve en:

$$\mathbf{Pf}(b,\otimes)\ ;\ \mathbf{Pf}(a\text{-}2*\otimes,\circledR);\ b := 3*\circledR$$

Empezando por la llamada externa, obtenemos:

$$\mathbf{Pf}(a\text{-}2*\mathbf{f}(b),\circledR);\ b := 3*\circledR$$

donde ahora debemos resolver la llamada a **f** presente dentro de la llamada a **Pf**, que resulta en:

$$\mathbf{Pf}(b,\otimes)\ ;\ \mathbf{Pf}(a\text{-}2*\otimes,\circledR);\ b := 3*\circledR$$

Note que ambas soluciones son equivalentes.

<u>Otro ejemplo</u>:

Sea la función $f(x)= x^2 + 3x - 2$, y su implementación como:

func F(x: real) $\rightarrow$ real
{Pre: true }
{Post: $F = x^2 + 3x - 2$ }

```
[
  var z : real;
  z := x^2 + 3*x -2

  >> z
]
```

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.

Sea PF el procedimiento asociado a F. Si queremos probar formalmente que se cumple:

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

deberíamos probar que se cumple:

{ P2 } **PF**(x1,y1) ; **Pf**(x2,y2); 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) Utilizando una función:

```
func exponenciación(b, e : int) → int
    { Pre: e ≥ 0 }
    { Post: exponenciación = b^e }
```

demuestre que los problemas TresExp y ExpSobreExp de los ejercicios 1) y 2) de la sección 4.3.2.1, pueden ser resueltos correctamente por las instrucciones:

```
resA := exponenciación(dato0, dato1);
resB := exponenciación(dato2, dato3);
resC := exponenciación(dato4, dato5)
```

y

```
r := exponenciación(a, exponenciación(b, c))
```

respectivamente.

2) 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. Haga las demostraciones de correctitud que correspondan.

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 $(a \geq 3) \rightarrow x := 4$ [] $(a \leq 3) \rightarrow x := 5$ **fi**

if $(a \geq 3) \rightarrow x := 4; b := a*2$ [] $(a < 3) \rightarrow a := 5$ **fi**

if $(a \geq 3) \rightarrow x := 4; b := a*2$ []
$(a < 3) \rightarrow$ **if** $(x \geq 3) \rightarrow x := 4$ [] $(x < 3) \rightarrow x := 5$ **fi;** $a := 5$
fi

if $((x \geq 3) \wedge p) \rightarrow x := 4$ [] $(\neg p) \rightarrow$ **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) $P \Rightarrow B_0 \vee B_1$ 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

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

$(b = |A|)(b := -A)$
$\equiv$ por sustitución
$(-A = |A|)$
$\equiv$ por aritmética
$(A \leq 0)$
$\Leftarrow$ por aritmética
$(A < 0)$
$\equiv$ por cálculo de predicados
verdad $\wedge (A < 0)$

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

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

$(b = |A|)(b := A)$
$\equiv$ por sustitución

$(A = |A|)$
≡ por aritmética
$(A \geq 0)$
≡ por cálculo de predicados
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 ≥ 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 ∈ ℜ) ∧ (A.x² + 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 ∧ (∀x: x ∈ ℜ: A.x² + B.x + C ≠ 0)) ∨
        (n=1 ∧ (∀x: x ∈ ℜ: A.x² + B.x + C = 0 ≡ x = x1)) ∨
        (n=2 ∧ (∀x: x ∈ ℜ: A.x² + B.x + C = 0 ≡ x = x1 ∨ x = x2)) ∨
        (n=3 ∧ (∀x: x ∈ ℜ: A.x² + 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 $(A=0 \wedge B=0 \wedge C\neq0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0)$ no es sólo condición suficiente sino además condición necesaria para que no exista solución. Formalmente tenemos:

$$(A=0 \wedge B=0 \wedge C\neq0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0) \equiv (\forall x: x \in \Re: A.x^2 + B.x + C \neq 0)$$

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

$$Q0': (n = 0 \wedge ((A=0 \wedge B=0 \wedge C\neq0) \vee (A \neq 0 \wedge B^2 - 4.A.C < 0)))$$

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:

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

Y podemos utilizar una propiedad similar a la anterior:

Si tenemos que:

$R \equiv (\exists \ sol1, sol2:: sol1 \neq sol2 \wedge (\ \forall x:: P(x) \equiv x = sol1 \vee x = sol2 \ ) \ )$

Entonces se cumple que:

$(\ \ \forall x: : P(x) \equiv x = d \vee \ \ x = e \ ) \wedge d \neq e \equiv \ \ R \wedge P(d) \wedge P(e) \wedge d \neq e$

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 $\wedge$ ((A=0 $\wedge$ B=0 $\wedge$ C$\neq$0) $\vee$ (A $\neq$ 0 $\wedge$ B^2 – 4.A.C < 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 $\wedge$ 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) $\vee$
  (n = 3 $\wedge$ A = 0 $\wedge$ B = 0 $\wedge$ 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 ($N \geq 0$) 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:

$B_0 \rightarrow S_0$

[] $B_1 \rightarrow S_1$
[] ...
[] $B_n \rightarrow S_n$

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:

(1)

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

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

   { suma = ( ∑i: 0 ≤ i < N : i² )  }
]
```

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!!

Definición formal de una instrucción iterativa:

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 $\rightarrow$ S **od** { Q }, es equivalente a demostrar que se cumple:

{ P }
if $\neg$B $\rightarrow$ **skip**
[] B $\rightarrow$ S; **do** B $\rightarrow$ S **od**
fi
{ Q }

Haciendo las anotaciones en la instrucción condicional, obtenemos:

{ P }
if $\neg$B $\rightarrow$ { P $\wedge$ $\neg$B} **skip** { Q }
[] B $\rightarrow$ { P $\wedge$ B} S; **do** B $\rightarrow$ S **od** { Q }
fi
{ Q }

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

{ P }
if $\neg$B $\rightarrow$ { P $\wedge$ $\neg$B} **skip** { Q }
[] B $\rightarrow$ { P $\wedge$ B} S { P }; **do** B $\rightarrow$ 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 = $(\sum j: 0 \leq j < i+1 : j^2) \wedge 0 \leq i+1 \leq N$ 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.

Demostración del Teorema de la Invariancia:

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

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

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.

5. Técnicas básicas de programación de procesos iterativos

En los problemas de programación que estudiaremos de ahora en adelante podemos encontrar un proceso iterativo para resolverlos. Por lo tanto, el diseño de invariantes adecuados es crucial para encontrar un programa que resuelva el problema.

En las secciones que siguen presentaremos varias formas de encontrar invariantes. Una forma de encontrar el invariante es determinando, como ya hemos visto, un proceso iterativo que resuelva el problema y capturar mediante un predicado la esencia de dicho proceso; esta forma es la más intuitiva y requiere de mucha creatividad. Sin embargo, una gran cantidad de problemas de programación pueden ser resueltos aplicando las técnicas que discutiremos en este capítulo, las cuales permiten **derivar** el programa a partir de la manipulación de las pre y post condiciones de una especificación formal de un programa.

Cualquiera sea la forma en que desarrollemos una instrucción iterativa para resolver un problema dado, sea utilizando la intuición o aplicando las técnicas que veremos en este capítulo, es conveniente seguir los siguientes pasos en el desarrollo y en este orden:

1) Determinar el Invariante y las guardias.
2) Inicialización: Establecer el invariante la primera vez. Es decir, establecer el valor inicial de las variables para que se cumpla el invariante.
3) Determinar la función de cota a partir de (1) y (2).
4) Desarrollar el trozo de programa que corresponde al cuerpo de cada guardia de manera que cumpla la especificación: {Invariante y Guardia} S { Invariante }. Este paso se puede descomponer en primero modificar la función de cota como última instrucción de S y en función de esto desarrollar el resto de S. La modificación de la función de cota es la que garantiza que la función decrece (o crece) estrictamente.

5.1. Técnica: Eliminar un predicado de una conjunción

Tenemos el problema siguiente: Calcular, utilizando sólo sumas, restas y multiplicaciones, el cociente y el resto de la división entera de A entre B, donde A y B son números enteros, $A \geq 0$ y $B > 0$.

La especificación formal es:

```
[  const  A, B: entero;
   var  q, r: entero;
   {  A ≥ 0   ∧   B > 0  }
    divmod
   {  q = A div B  ∧  r = A mod B  }
]
```

Usando la definición de div y mod, obtenemos una especificación mas refinada:

```
[  const  A, B: entero;
```

```
  var  q, r: entero;
  { A ≥ 0  ∧  B > 0 }
   divmod
  { A = q * B + r  ∧ 0 ≤ r  ∧  r < B }
]
```

La postcondición puede ser manipulada de la siguiente forma:

$$
\begin{aligned}
& A = q * B + r \ \wedge\ 0 \le r \ \wedge\ r < B \\
\equiv\ & \text{aritmética (despejando r)} \\
& r = A - q * B \ \wedge\ 0 \le r \ \wedge\ r < B \\
\equiv\ & \text{lógica (sustitución de iguales)} \\
& r = A - q * B \ \wedge\ 0 \le A - q * B \ \wedge\ A - q * B < B \\
\equiv\ & \text{aritmética (sumando B en la segunda fórmula)} \\
& r = A - q * B \ \wedge\ B \le A - q * B + B \ \wedge\ A - q * B < B \\
\equiv\ & \text{aritmética (sacando factor común)} \\
& r = A - q * B \ \wedge\ B \le A - (q - 1) * B \ \wedge\ A - q * B < B
\end{aligned}
$$

Entonces, encontrar un q y un r que satisfacen la postcondición anterior es equivalente a hallar un q (entero no negativo) tal que $0 \le A - q*B < B$ y esto a su vez es equivalente a hallar un entero no negativo q tal que $A - q*B < B$ y $A - (q-1)*B \ge B$. Y r será igual a $A - q*B$. Es decir, buscar un q tal que al restarle a A la cantidad $(q-1)*B$, el resultado (llamémoslo **resta**) es mayor o igual que B, pero al restarle de nuevo B a **resta** ($A - q*B = A - (q-1)*B - B$), el resultado da menor que B.

Por lo tanto nuestro problema lo podemos especificar también como:

```
[ const  A, B: entero;
  var  q, r: entero;
  { A ≥ 0  ∧  B > 0 }
   divmod
  { A – q*B < B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Si existe un q que satisface la postcondición entonces éste sólo puede tomar valores entre 0 y A, ambos inclusive. Por lo que bastará con recorrer los números naturales entre 0 y A para encontrar dicho número. Veamos por qué.

El primer predicado de la postcondición, $A - q*B < B$, implica que 0 es cota inferior de q:

$$
\begin{aligned}
& A - q*B < B \\
\equiv\ & \text{aritmética (sumando q*B a ambos lados y sacando factor común)} \\
& A < (q+1)*B \\
\Rightarrow\ & \text{como } A \ge 0 \text{ por precondición}
\end{aligned}
$$

$0 < (q+1)*B$

$\equiv$ aritmética (pues $B > 0$ por precondición)

$0 < (q+1)$

$\equiv$ restando 1 a ambos lados

$-1 < q$

$\equiv$ aritmética (q es entero y mayor estricto que -1)

$0 \leq q$

El segundo predicado en la postcondición, $A - (q-1)*B \geq B$, implica que q tiene como cota superior a A:

$A - (q-1)*B \geq B$

$\equiv$ aritmética (restando B a ambos lados y sacando factor común)

$A - q*B \geq 0$

$\equiv$ aritmética

$A \geq q*B$

$\Rightarrow$ como $B > 0$ (precondición), se tiene que $B \geq 1$ y así $q*B \geq q$

$A \geq q$

Por lo tanto hemos demostrado que:

$$A - q*B < B \wedge A - (q-1)*B \geq B \Rightarrow 0 \leq q \wedge q \leq A$$

Un proceso iterativo que resuelve el problema es recorrer con la variable q los números naturales desde 0 hasta A (vimos que q no puede ser mayor que A y alcanza el valor de A con B=1 y r=0). Y partiendo de q=0, mientras $A - q*B$ sea mayor o igual que B, sumar 1 a q. El número de iteraciones del proceso iterativo está acotado por A.

Note que al concluir el proceso iterativo tendremos $q = A \text{ div } B$. Y podemos entonces asignar a r el valor $A - q*B$. Note también que al comienzo de cada iteración (incluyendo la última, cuando la condición de continuación no se satisface) se cumple que $A - (q-1)*B \geq B$, y este captura la esencia del proceso.

Así nuestro programa sería:

```
[ const  A, B: entero;
  var  q, r: entero;
  { A ≥ 0  ∧  B > 0 }
  q := 0;
  do A – q*B ≥ B → q := q + 1
  od;
   r := A – q*B
  { A – q*B < B ∧ A – (q-1)*B ≥ B ∧ r = A – q*B }
]
```

Ejercicio: muestre que el programa anterior es correcto utilizando el invariante $A - (q-1)*B \geq B$ y la función de cota creciente q (o función de cota decreciente A-q).

Para el problema anterior podemos utilizar una técnica para derivar el programa, conocida como **la técnica de eliminación de un predicado cuando la postcondición se expresa como una conjunción de predicados.** Esta técnica expresa lo siguiente:

Cuando la postcondición R es de la forma $P \wedge Q$, uno puede tratar de tomar uno de los predicados como el invariante y el otro como la negación de la guardia de la instrucción iterativa, es decir:

$\{ P \}$ do $\neg Q \rightarrow S$ od $\{ P \wedge Q \}$

```
[ const  A, B: entero;
  var  q, r: entero;
  { A ≥ 0  ∧  B > 0 }
   divmod
  { R: A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Tomemos como invariante a P: $A - (q-1)*B \geq B \wedge r = A - q*B$, y a $A - q*B \geq B$ como guardia. Lo cual resulta en un programa de la forma:

$\{ P \}$ do $A - q*B \geq B \rightarrow S$ od $\{ R \}$

El invariante puede ser establecido inicialmente mediante las asignaciones q := 0 y r := A. Como P implica $A - q*B \geq 0$ y debe decrecer en cada iteración hasta llegar a $A - q*B < B$, podemos tomar a $A - q*B$ como función de cota. En cada iteración podemos aumentar en 1 a q para que decrezca $A - q*B$ y para mantener el invariante habría que reducir en B a r. Esto lleva al siguiente programa **divmod:**

```
[ const  A, B: entero;
  var  q, r: entero;
  { A ≥ 0  ∧  B > 0 }
  q := 0;  r := A;
  do A – q*B  ≥  B →  r, q :=  r – B, q := q +1
  od
  { R: A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Note además, que al ser $r = A - q*B$ un invariante, podemos reemplazar la expresión $A - q*B$ en el programa anterior por r, lo cual mejora la eficiencia del programa al no tener que calcular cada vez la guardia. Y el invariante queda: $r \geq 0 \wedge r = A - q*B$. quedando el programa con anotaciones siguiente:

```
[ const  A, B: entero;
```

```
  var  q, r: entero;
  {  A ≥ 0  ∧  B > 0  }
  q := 0;  r := A;
  { Invariante P: r  ≥  0  ∧ r = A – q*B, demo 0, función de cota decreciente: r }
  do  r  ≥  B  →  { P ∧  r  ≥  B }  r, q :=  r – B; q := q +1  { P, demo 1 }
  od
  {  R: r <  B  ∧  r  ≥  0  ∧  r = A – q*B, demo 2, terminación: demo 3 }
]
```

Hagamos las demostraciones demo 0, demo1, demo 2 y demo 3 para demostrar la correctitud del programa anterior.

Demo 0:

Mostremos que se cumple:

```
  {  A ≥ 0  ∧  B > 0  }
  q := 0;  r := A
  { r  ≥  0  ∧ r = A – q*B}
```

En efecto:

```
        ( r  ≥  0  ∧ r = A – q*B ) (r := A) (q := 0)
≡    sustitución
        A  ≥  0  ∧ A = A – 0*B
≡ aritmética
        A  ≥  0 ∧ A = A
≡ aritmética
        A ≥ 0
⇐       lógica ( p ∧ q ⇒ p )
        A ≥ 0  ∧  B > 0
```

Demo 1:

Demostremos que se cumple { P ∧ r ≥ B } r, q := r – B, q +1 { P }

```
        P( r, q :=  r – B, q+1)
≡ sustitución
        r - B  ≥  0  ∧ r – B  = A – (q+1)*B
≡ aritmética
        r  ≥  B  ∧ r  = A – q*B
⇐ por lógica ( p ∧ q ⇒ p )
        r  ≥  B  ∧ r  = A – q*B ∧ r  ≥  0
```

Demo 2:

Es directa.

Demo 3 (terminación):

Dada la función de cota r, sabemos por el invariante que $r \geq 0$ al comienzo de cada iteración.

Debemos mostrar ahora que si se tiene r = C al comienzo de una iteración y se cumple la guardia entonces al concluir la iteración se tendrá r < C. En efecto, supongamos que r = C. La instrucción correspondiente a la guardia resta B a r, por lo que al concluir la instrucción r será igual a C-B, y como B es positivo, el nuevo valor de r será menor estricto que el original.

Hemos podido aplicar directamente la **técnica de eliminación de un predicado de la conjunción**, a la especificación:

[const A, B: entero;
  var q, r: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  divmod
  { R: $A = q*B + r \wedge r \geq 0 \wedge r < B$ }
]

Tomando como invariante a P: $A = q*B + r \wedge r \geq 0$, y $r \geq B$ como la guardia. Lo cual resulta en un programa de la forma:

{ P } do $r \geq B \rightarrow$ S od { R }

El invariante puede ser establecido inicialmente mediante las asignaciones q := 0 y r := A. Como P implica $r \geq 0$ y r debe decrecer en cada iteración hasta llegar a $r < B$, podemos tomar a r como función de cota. En cada iteración podemos reducir r en B y para mantener el invariante habría que aumentar q en 1. Esto lleva al siguiente programa **divmod:**

[const A, B: entero;
  var q, r: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  q := 0; r := A;
  do $r \geq B \rightarrow$ r, q := r – B, q+1
  od
  { $A = q*B + r \ \wedge \ r \geq 0 \wedge r < B$ }
]

El programa anterior es exactamente el mismo que obtuvimos antes.

Hemos podido tomar como invariante cualquier combinación de conjunciones y tratar de derivar el programa. Es posible que no tengamos éxito con algunas combinaciones.

Ejercicio: Utilice la técnica anterior tomando como invariante a: $A - q*B < B$, para derivar el programa dado por la especificación siguiente (este programa determina el cociente q de la división entera de A entre B y sólo debe usar sumas y restas):

[const A, B: entero;
  var q: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  división entera
  { $A - (q-1)*B \geq B \ \wedge \ A - q*B < B$ }
]

Ejemplo: Cálculo de la raíz cuadrada entera de un número entero no negativo.

Problema: Dado un número entero no negativo N, se quiere determinar la raíz cuadrada entera de N, es decir, el mayor número entero no negativo que multiplicado por el mismo es menor o igual a N.

Este problema es "del mismo tipo" que el anterior, en el sentido que hay que recorrer el conjunto de los números naturales, partiendo desde cero hasta un cierto número, hasta encontrar un número x que cumpla una determinada propiedad P(x) y el sucesor del número, x+1, no cumpla la propiedad (se cumple $\neg P(x+1)$). En el ejemplo anterior (primera versión de desarrollo) la propiedad era P(q): $A - (q-1)*B \geq B$.

En el ejemplo que nos ocupa, queremos hallar un número entero no negativo x tal que $x^2 \leq N$ y $(x+1)^2 > N$, donde la propiedad P(x) es $x^2 \leq N$.

La especificación formal es:

[const N: entero;
  var x: entero;
  { $N \geq 0$ }
  raíz cuadrada
  { $x^2 \leq N \ \wedge \ (x+1)^2 > N$ }
]

Como N es no negativo, podemos recorrer los números naturales partiendo desde cero hasta encontrar el número x que cumpla $x^2 \leq N$ y $(x+1)^2 > N$. Note que x nunca podrá exceder a N (si $x > N$ entonces por ser $N \geq 0$ se tiene que $x \geq 1$ y así $x^2 \geq x > N$, es decir, x no cumple con $x^2 \leq N$). Por lo tanto en el proceso iterativo que consiste en recorrer los números naturales partiendo desde cero hasta encontrar el número buscado siempre se ha de cumplir el predicado $x^2 \leq N$ (este sería el invariante) y la condición de continuación del proceso iterativo deberá ser $(x+1)^2 \leq N$. Note que $x = 0$ satisface inicialmente el invariante.

Como $x^2 \leq N$ es equivalente a $N - x^2 \geq 0$, podemos tomar como función de cota decreciente a $N - x^2$, sin embargo, $N - x^2$ decrece con un incremento de x si y sólo si $x \geq 0$, y esto no puede ser inferido del invariante y la guardia: $x^2 \leq N \wedge (x+1)^2 \leq N$. Por lo tanto debemos reforzar el invariante a: $x \geq 0 \wedge x^2 \leq N$.

Por lo tanto el programa es:

```
[ const  N: entero;
  var x: entero;
  { N ≥ 0 }
  x := 0;
  { Invariante P: x ≥ 0 ∧ x² ≤ N, demo 0, función de cota decreciente: N – x² }
  do  (x+1)*(x+1) ≤ N  →  x  :=  x+1
  od
  {  x² ≤ N  ∧  (x+1)² > N  }
]
```

Note que otra solución iterativa al problema de hallar un entero x que cumpla P(x) y $\neg$P(x+1), es partir de un número entero x que cumpla $\neg$P(x+1) y luego ir disminuyendo x en 1 hasta encontrar un x que cumpla $P(x) \wedge \neg P(x+1)$.

Ejercicios:

1) demuestre la correctitud del programa anterior.
2) Ejercicios página 56 del Kaldewaij.
3) Aplique **técnica de eliminación de un predicado de la conjunción** a la especificación:

```
[ const  N: entero;
  var x: entero;
  { N ≥ 0 }
   raíz cuadrada
  {  x² ≤ N  ∧  (x+1)² > N  }
]
```

utilizando como invariante a $x^2 \leq N$, luego utilizando como invariante a $(x+1)^2 > N$

5.2. Técnica: reemplazo de constantes por variables

Problema: queremos calcular A^B, con B entero no negativo, donde, por definición, $A^0 = 1$, para todo A (incluyendo 0).

Este problema se especifica formalmente como sigue:

```
[  const  A, B:entero;
  var r: entero;
  {  B ≥ 0 }
```

exponenciación
{ r = A^B }
]

Un proceso iterativo que resuelve este problema es: en cada iteración multiplicar a r (partiendo de r = 1) por A y guardar el resultado en r. Más formalmente, al comienzo de la iteración i (comenzando desde la iteración 0) la variable r contendrá A^i, por lo que un invariante sería r = A^i. Note que si al comienzo de la iteración 0 (i=0) se deberá tener r =1. El esquema del programa con la fase de inicialización sería:

r , i := 1, 0;
{ Invariante: r = A^i}
do B $\rightarrow$ S
od
Como el número de la iteración i cumple con $0 \leq i \leq B$, podemos incorporar este predicado al invariante. Por otro lado, la condición de continuación del proceso iterativo debe ser i < B.

r , i := 1, 0;
{ Invariante: r = $A^i \wedge 0 \leq i \leq B$}
do i < B $\rightarrow$ S
od

Note que para que al comienzo de la iteración i+1 la variable r contenga A^{i+1}, basta con asignar r*A a r en la iteración i, e incrementar en 1 la variable i para que refleje la siguiente iteración:

r , i := 1, 0;
{ Invariante: r = $A^i \wedge 0 \leq i \leq B$}
do i < B $\rightarrow$ r, i := r*A, i+1
od

Por lo tanto, al comienzo de la iteración B la variable r contendrá el resultado y debe concluir el proceso iterativo. Una función de cota estrictamente creciente entre dos iteraciones sucesivas es i (el número de la iteración), la cual está acotada superiormente por B. Podemos tomar también como función de cota a B-i (estrictamente decreciente entre dos iteraciones sucesivas) y acotada inferiormente por 0.

El programa es:

[const A, B:entero;
var r, i: entero;
{ P: $B \geq 0$ }
r, i := 1, 0;
{Invariante I: r = $A^i \wedge 0 \leq i \leq B$, función de cota creciente: i }
do i < B $\rightarrow$ r, i := r*A, i+1

{ Q: $r = A^B$ }
]

Para demostrar la correctitud del programa anterior, hay que demostrar:

1) Se cumple: { P } r, i := 1, 0; { I }
2) Se cumple: { $I \wedge i < B$ } r := r*A; i := i+1{ I }
3) [$I \wedge i \geq B \Rightarrow Q$]
4) Se cumple: { $I \wedge i = C$ } r := r*A; i := i+1 { $i < C$ }

Mostremos (2):

I(r, i := r*A, i+1)
≡ sustitución
$r*A = A^{i+1} \wedge 0 \leq i+1 \leq B$

Suponiendo que se cumple $I \wedge i < B$, es decir, $r = A^i \wedge 0 \leq i \leq B \wedge i < B$, tenemos que multiplicando por A en ambos lados de $r = A^i$ tenemos $r*A = A^{i+1}$. Por otro lado al ser $i \geq 0$ entonces se tiene que $i+1 \geq 0$ y al ser $i < B$ se tiene que $i+1 \leq B$.

La **técnica de reemplazo de constantes por variables** nos permite derivar el mismo programa anterior estableciendo un invariante a partir de la postcondición mediante el reemplazo de una constante de la postcondición por una variable nueva. Las posibilidades son las siguientes:

$$r = x^B, \quad r = A^x, \quad r = x^y$$

Utilicemos como invariante a

P_0: $r = A^x$

Entonces $P_0 \wedge x = B$ implica la postcondición, y P_0 puede ser establecida con la inicialización r, x :=1, 0.

Siempre es conveniente acotar a las variables nuevas que introduzcamos. Podemos fijar una cota superior para x, y agregar el predicado siguiente al invariante:

P_1: $x \leq B$

Esto lleva al esquema de programa:

$$R, x := 1, 0 \ \{ P_0 \wedge P_1 \} ; \textbf{do } x \neq B \rightarrow S \textbf{ od } \{ r = A^B \}$$

Como x inicialmente es cero y deseamos llegar a x = B, debemos investigar el efecto de incrementar en 1 a x como última instrucción de S:

$(P_0 \wedge P_1)(x := x+1)$

$\equiv$ por sustitución

$r = A^{x+1} \wedge x + 1 \leq B$

Si a S lo representamos por la secuenciación S1; x:= x+1, deberíamos tener:

(1) $\{ P_0 \wedge P_1 \wedge x \neq B\}$ S1 $\{ r = A^{x+1} \wedge x + 1 \leq B \}$; x := x+1 $\{ P_0 \wedge P_1 \}$

Si P_0 se cumple tenemos que $A^{x+1} = A^* A^x = A * r$, por lo que S1 sería la instrucción r := A * r, además, $P_1 \wedge x \neq B$ no son afectados por la asignación a r e implican $x < B$, y esto último implica $x+1 \leq B$.

Así, el programa exponenciación es:

```
[ const  A, B:entero;
  var r, x: entero;
  { A ≥ 0 ∧ B ≥ 0 }
  r, i := 1, 0;
  {Invariante: r = A^x ∧ x ≤B, función de cota decreciente: B-x }
  do x ≠ B → r, x := A*r, x+1
  { r = A^B }
]
```

Ejercicios:

1) Hacer un programa para los ejercicios 12, 13, 15, 16, 17, 24, 25, 26, 30 de la sección 1 del problemario de Michel Cunto.
2) Dibujar N círculos concéntricos con la máquina de trazados, con centro (0,0) y los radios siguen la progresión aritmética 1+2*i, para los círculos 0,1,...N-1

Problema: sumar los elementos de una secuencia de enteros de largo N.

La especificación es:

```
[ const  N: entero;
  const f: secuencia de enteros;
  var  x: entero;
  { |f| = N ∧ N ≥ 0 }
  suma
  { x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

En este ejemplo utilizaremos el tipo de datos “secuencia de enteros” y expresaremos la solución del problema en términos de las operaciones de observación de los elementos de una secuencia.

Un proceso iterativo que resuelve este problema es muy parecido al que describimos en la sección 4.5, la suma de los cuadrados de los primeros N números naturales. El proceso consiste en ir acumulando en la variable x la suma de los elementos de la secuencia. Al comienzo de la iteración i (comenzando con la iteración 0) se han sumado los elementos de la secuencia entre 0 y i-1. El proceso termina en la iteración N. Por lo tanto, un invariante para este proceso es: $(x = (\sum j: 0 \le j < i : f[j])) \wedge (0 \le i \le N)$. En cada iteración, menos en la última, sumamos el elemento f[i] a la variable x. Sabemos que $(\sum i: 0 \le i < 0 : f[i])$ es igual a cero (la suma de cero números es cero), y entonces inicialmente la variable x es cero. El número de la iteración i sirve como función de cota creciente.

El programa sería:

```
[ const  N: entero;
  const f: secuencia de enteros;
  var  x,i: entero;
  {  |f| = N  ∧  N  ≥  0 }
  x , i := 0, 0;
  { Invariante I: ( x = ( ∑j: 0 ≤ j < i : f [j] ) )  ∧  (0 ≤ i ≤ N),
                                                  función de cota creciente: i }
  do  i < N  →  x, i := x +f[i], i+1  od
  {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

Ejercicio: demuestre la correctitud del programa anterior.

Podemos aplicar la **técnica de reemplazo de constantes por variables** para derivar un invariante y obtener el mismo programa anterior. Reemplazamos una constante de la postcondición $x = (\sum j: 0 \le j < N : f[j])$ por una variable nueva. El cuantificador que aparece en esta postcondición posee dos constantes: 0 y N. Reemplacemos N por una variable i y propongamos como invariante a:

$$P_0: \quad x = (\sum j: 0 \le j < i : f[j])$$

Por lo tanto $P_0 \wedge i=N$ implica la postcondición. La suma sobre un rango vacío es cero, por lo que inicialmente se puede establecer P_0 con la instrucción x, i := 0, 0. El programa que buscamos tiene el esquema siguiente:

$$x, i := 0, 0;\ \text{do } i \neq N \rightarrow S \text{ od}$$

Debemos encontrar S. Supongamos que se cumple $P_0 \wedge i \neq N$, es decir, se cumple $x = (\sum j: 0 \le j < i : f[j]) \wedge i \neq N$. Este predicado refleja el estado de las variables justo antes de ejecutarse S. Como el interés es incrementar i hasta llegar a N, veamos lo que significa un incremento de i en 1 en la expresión que refleja la suma en P_0:

$(\sum j: 0 \le j < i+1 : f[j])$

= siempre y cuando $i \ge 0$ podemos separar (habría que agregar este predicado a P_0

como invariante)
$(\sum j: 0 \le j < i : f[j]) + f[i]$
= como $x = (\sum j: 0 \le j < i : f[j])$ por P_0
$x + f[i]$

Por lo tanto basta con asignar a x el valor de la expresión x + f[i] para que se cumpla $P_0(i := i+1)$, es decir, $x = (\sum j: 0 \le j < i+1 : f[j])$. Tenemos entonces que se cumple:

$$\{ P_0 \wedge 0 \le i \wedge i \neq N \}\ x := x+f[i]\ \{ P_0(i := i+1) \}\ ;i := i+1\ \{ P_0 \}$$

Una función de cota apropiada es N-i (también sirve i, creciente estricta). Esta sería estrictamente decreciente pues i aumenta en 1 en cada iteración. Sin embargo para la prueba de terminación necesitaremos que siempre se cumpla $i \le N$, lo que garantiza que la función de cota está acotada inferiormente por cero.

Por lo tanto, haber hecho el reemplazo de una constante por una variable en la postcondición nos llevó a encontrar los siguientes invariantes:

P_0: $x = (\sum j: 0 \le j < i : f[j])$
P_1: $0 \le i \le N$

Finalmente el programa es:

```
[ const  N: entero;
  const f: secuencia de enteros;
  var  x,i: entero;
  { |f| = N ∧ N ≥ 0 }
  x , i := 0, 0;
  { Invariante I: ( x = ( ∑j: 0 ≤ j < i : f [j] ) ) ∧ (0 ≤ i ≤ N), cota: i }
  do  i ≠ N → x, i := x +f[i], i+1  od
  {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

Ejercicios:

1) Aplicando la técnica de derivación de reemplazo de constantes por variables, reemplace la constante 0 por n en la postcondición del ejemplo anterior con el objeto de hallar un invariante y derive el programa correspondiente.

5.3. Técnica: fortalecimiento de invariantes

Problema: queremos determinar el N-ésimo número de Fibonacci, fib(N). Los números de Fibonacci se definen de la siguiente forma:

fib(0) = 0, fib(1) = 1, y fib(n+2) = fib(n) + fib(n+1) para $n \ge 0$.

La especificación formal del programa sería:

[const N: entero;
  var x: entero;
  { $N \geq 0$ }
  Fibonacci
  { x = fib(N) }
]

Aplicando la técnica de reemplazo de una constante por una variable en la postcondición, si reemplazamos a N por i, podemos proponer como invariante a "x = fib(i)". Este invariante nos dice que al comienzo de la iteración i se tiene x = fib(i). Vemos que x = fib(i) $\wedge$ i = N implican la postcondición, por lo que la guardia sería $i \neq N$. El invariante se puede establecer inicialmente asignando a x el valor fib(0) (= 0) y asignar 0 a i . Más aún, tenemos como invariante también a $0 \leq i \leq N$. Y una función de cota decreciente: N-i.

El esquema del programa es:

x , i := 0, 0;
do $i \neq N \rightarrow$ S od

Determinemos S. Note que al comienzo de la iteración i+1 la variable x debe contener fib(i+1). Suponiendo que al comienzo de la iteración i se cumple x = fib(i) $\wedge$ $0 \leq i \leq N \wedge i \neq$ N (es decir, el invariante y la guardia), no tenemos forma de calcular fib(i+1) (que será el valor de x al concluir la iteración i) a partir sólo de x y de i. Por lo que introducimos una nueva variable z que al comienzo de la iteración i contenga fib(i+1). De esta forma **fortalecemos nuestro invariante (aquí estamos aplicando la técnica de fortalecimiento del invariante)** y éste será:

$$x = fib(i) \wedge z = fib(i+1) \wedge 0 \leq i \leq N$$

Que puede ser establecido inicialmente asignando 0, 1, 0 a x, z, i respectivamente. El nuevo esquema de programa sería:

x , z, i := 0, 1, 0;
do $i \neq N \rightarrow$ S od

Para que se mantenga el invariante, como estamos analizando un proceso iterativo, al comienzo de la iteración i+1 la variable x deberá contener fib(i+1) y la variable z deberá contener fib(i+2). En la iteración i se deberá colocar en x el valor fib(i+1) (= valor inicial de z al comienzo de la iteración i) y en la variable z se deberá asignar el valor fib(i+2) (= fib(i) + fib(i+1) = valor inicial de x + valor inicial de z al comienzo de la iteración i).

Por lo que el programa sería:

[const N: entero;

```
  var x, y: entero;
  { N ≥ 0 }
  x , z, i := 0, 1, 0;
  do i ≠ N → "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales
                de x, z";
                i := i + 1
  od
  { x = fib(N) }
]
```

Otra manera de razonar para deducir el trozo de programa "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales de x, z", sería como sigue:

Sean P_0: x = fib(i) , P_1: z = fib(i+1), R: $0 \le i \le N$. Suponiendo que se cumple $P_0 \wedge P_1 \wedge R \wedge i \neq N$, un incremento de i en 1 nos debería llevar a que se cumpla $(P_0 \wedge P_1 \wedge R)$ (i := i+1) y esto es equivalente a $x = fib(i+1) \wedge z = fib(i+2) \wedge 0 \le i+1 \le N$. Asignando el valor de z a x queda establecido x = fib(i+1). El predicado $0 \le i+1 \le N$ queda directamente establecido pues se tiene $i \neq N \wedge 0 \le i \le N$. El predicado z = fib(i+2) queda establecido por:

```
        fib(i+2)
=  definición de fib
        fib(i) + fib(i+1)
= por P0 y P1
        x + z
```

Por lo tanto, el trozo de programa "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales de x, z" lo podemos implementar con una asignación múltiple x, z := z, x + z, o introduciendo una nueva variable t que conserve el valor original de una de las dos variables x, z: t := x; x:= z; z:= t + z

Finalmente el programa con sus anotaciones sería:

```
[ const N: entero;
  var x, z, t: entero;
  { N ≥ 0 }
  x , y, i := 0, 1, 0;
  { Invariante: x = fib(i) ∧ z = fib(i+1) ∧ 0 ≤ i ≤ N, demo 0;
     función de cota decreciente: N-i }
  do i ≠ N → { x = fib(i) ∧ 0 ≤ i ≤ N ∧ i ≠ N}
                x, z := z, x + z;
                i := i+1
                { x = fib(i) ∧ 0 ≤ i ≤ N, demo 1 }
  od
  { x = fib(N), demo 2, terminación: demo 3 }
]
```

Ejercicios:

1) Hacer un programa que calcule (max p: $0 \leq p \leq N$: ($\sum$j: $p \leq j < N$: s[j])), para $N \geq 0$ y s una secuencia de largo N. Ayuda: trate de simplificar la expresión y aplicar las técnicas de reemplazo de constantes por variables y aplicar la regla de fortalecimiento del invariante reemplazando en la postcondición la constante 0 por una variable.

2) Desarrolle un programa correcto que resuelva los siguientes problemas (trate de utilizar, de ser posible, las técnicas dadas de derivación):

   A. Calcular ($\sum$i: $0 \leq i < N$: i^3).
   B. Calcular ($\sum$i: $0 \leq i < N$: s[i]). Donde s es una secuencia de largo N.
   C. Calcular ($\prod$i: $0 \leq i < N$: 1/(i+1)).
   D. Calcular el mayor elemento de una secuencia s de números enteros de largo n.

Páginas 62 y 71 de Kaldewaij, reemplazando la palabra “arreglo” por “secuencia”.

6. Arreglos

En este capítulo definimos el tipo **arreglo**. La mayoría de los lenguajes de programación (JAVA, C, PASCAL, etc.) poseen el tipo arreglo. Como veremos, los arreglos permiten implementar, representar y manipular de una manera muy conveniente al tipo abstracto de dato **secuencia**, en particular permiten implementar de manera sencilla los cambios de valores que pueda tener una variable tipo secuencia.

6.1. Definición del tipo arreglo

Un conjunto finito de enteros consecutivos se denomina un ***segmento***. Por ejemplo {0, 1, 2} es un segmento y lo denotamos por **[0..3)**. En general, dados dos enteros p, q con $p \leq q$, el segmento de los enteros i que satisfacen $p \leq i < q$, lo denotamos por [p..q). Note que p = q implica que el segmento [p..q) es el conjunto vacío. También note que el número de elementos del segmento [p..q) es q-p. Note además que [2..1) no es un segmento.

Un ***arreglo*** es una función parcial de los enteros en cualquiera de los tipos básicos ya vistos (entero, real, carácter, booleano) o el mismo tipo arreglo, y cuyo dominio es un segmento [p..q), para p, q dados. Note que si p=0 entonces el arreglo no es más que una secuencia. Por lo que usaremos la misma notación de secuencias para denotar la aplicación de la función en un elemento del segmento, por ejemplo si b es un arreglo con dominio [0..3) entonces b es una secuencia de tres elementos y b[2] representa la imagen de 2 según b, es decir, el último elemento de la secuencia. Decimos que b[0] es el primer elemento del arreglo b, b[1] es el segundo elemento del arreglo b, etc. Un arreglo con dominio [p..p) diremos que es un arreglo sin elementos, es vacío.

Si b es un arreglo con dominio [0..2) y rango los números enteros, y cuyos elementos tienen los valores b[0]=-4, b[1]=-5, entonces el valor del arreglo lo denotamos por b = <-4, -5>.

Decimos que el tipo arreglo es un tipo ***estructurado***, lo cual significa que los valores del tipo vienen definidos por una estructura sobre valores de otros tipos (en este caso una función de enteros a otro conjunto).

Tradicionalmente los arreglos han sido vistos como un conjunto de variables indexadas e independientes que comparten un mismo nombre: el arreglo b con dominio [0..3) corresponde a las variables b[0], b[1] y b[2] (ver figura 1). Sin embargo, ver a los arreglos como funciones nos ayudará a manejar la formalidad cuando demostremos correctitud de programas.

En nuestro pseudolenguaje declararemos un arreglo b de la forma siguiente:

arreglo [p..q) de <tipo>

donde <tipo> puede ser cualquier tipo de nuestro pseudolenguaje, inclusive el tipo arreglo mismo. Tanto p como q representan expresiones de tipo entero cuyos valores satisfacen $p \leq q$.

Ejemplo de declaraciones de arreglos:

1) var b: arreglo [0..N) de entero; (con N ≥ 0)
2) var f: arreglo [2..3) de entero;
3) var a: arreglo [1..6) de booleano;

En la figura 7 vemos la representación gráfica de un arreglo, donde cada casilla corresponde a una localidad de memoria, es decir, b[i] es una variable.

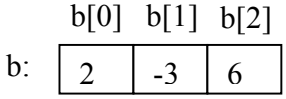


Figura 7

Si b es un arreglo con dominio [p..q) y p ≤ i ≤ j ≤ q, denotamos por b[i..j) los elementos del arreglo b correspondientes a los índices en el segmento [i..j) y decimos que es el segmento de b entre i y j. Note que, por ejemplo, b[p,p) (la secuencia vacía) es un segmento de b.

El tipo de los elementos de un arreglo puede ser un arreglo:

b: arreglo [p1..q1) de arreglo [p2..q2) de <tipo>;

Para abreviar la notación colocamos:

b: arreglo [p1..q1)×[p2..q2) de <tipo>;

Decimos que b es un arreglo de dimensión 2 de elementos de tipo <tipo>. Note también que b es un arreglo de dimensión 1 de elementos de tipo "arreglo [p2..q2) de <tipo>". Y vemos que b será una función del producto cartesiano [p1..q1)×[p2..q2) en el conjunto de valores de <tipo>. Vemos que b[i], p1 ≤ i < q1, es un arreglo cuyos elementos son de tipo arreglo [p2..q2) de <tipo>. Y b[i][j] será el elemento j del arreglo b[i]

Por ejemplo si b es un arreglo con dominio [0..2)×[1..4) en los enteros, y b=<<-4, -5, 2>, <5, 0, -3>>, entonces b[0][2] = -5.

Ejemplo de uso de arreglos:

El problema ya visto "sumar los elementos de una secuencia de enteros de largo N" puede ser implementado con el tipo arreglo, obteniéndose el programa siguiente:

```
[  const  N: entero;
   const f: arreglo [0..N) de entero;
   var  x,i: entero;
   {  N  ≥  0 }
```

```
  x := 0;
  i := 0;
  { Invariante I: ( x = ( ∑j: 0 ≤ j < i : f [j] ) )  ∧  (0 ≤ i ≤ N), función de cota creciente: i }
  do  i < N  →  x := x +f[i]; i := i+1  od
  {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

Veamos un nuevo problema donde utilizamos arreglos.

Problema: Dado un arreglo A de enteros con dominio [0,N), N ≥ 0, deseamos calcular el segmento A[p..q) de A cuya suma de sus elementos sea máxima entre todos los segmentos de A.

Una especificación formal de este problema es:

```
[  const N: entero;
   const A: arreglo [0..N) de entero;  {N ≥ 0}
   var suma: entero;
   { verdad }
   maxsegsum
   { suma = (max p,q: 0 ≤ p ≤ q ≤ N : (∑ j: p ≤ j < q : A[j]))  }
]
```

Para simplificar la notación, definimos, para $0 \le p \le q \le N$:

S(p,q): $(\sum j: p \le j < q : A[j])$

Una estrategia para resolver este problema es la siguiente:

Supongamos que N > 0 y hemos resuelto el mismo problema pero para el segmento de A[0..N-1), es decir, conocemos (max p,q: $0 \le p \le q \le N-1$: S(p,q)), que denotaremos por $suma_{N-1}$.

Por lo tanto (max p,q: $0 \le p \le q \le N$: S(p,q)) es el máximo valor entre $suma_{N-1}$ y la suma de cada uno de los segmentos A[j..N), para $0 \le j \le N$, esta suma es S(j,N). Esto se debe a que:

```
        (max p,q: 0 ≤ p ≤ q ≤ N : S(p,q))
=  como N > 0 existe un término que separar
        max ((max p,q: 0 ≤ p ≤ q ≤ N-1 : S(p,q)),  (max p: 0 ≤ p ≤  N : S(p,N))
=  por definición
        max (suma_{N-1},  (max p: 0 ≤ p ≤  N : S(p,N)) )
```

Note que $suma_{N-1}$ lo podemos expresar de igual forma como: max ($suma_{N-2}$, (max p: $0 \le p \le N-1$: S(p,N-1))). Por lo tanto el proceso iterativo que resuelve nuestro problema original es: al comienzo de la iteración i (comenzando desde 0) la variable **suma** contiene el valor

(max p,q: $0 \leq p \leq q \leq i$: S(p,q)), este sería nuestro invariante. Inicialmente podemos establecer el invariante con suma := 0; i:= 0 (la suma del segmento máximo de un arreglo vacío es cero). Cuando llegamos al comienzo de la iteración N hemos resuelto el problema. Por lo que otro invariante es $0 \leq i \leq N$. Y i (ó N-i) sirve como función de cota. Por lo tanto el esquema de nuestro programa sería:

suma := 0;
i := 0;
{ Invariante (suma = (max p,q: $0 \leq p \leq q \leq i$: S(p,q))) $\wedge$ $0 \leq i \leq N$ }
do i<N $\rightarrow$ *Prog* od

Del análisis hecho anteriormente *Prog* debe consistir en asignar a **suma** el máximo entre el valor de la variable **suma** al comienzo de la iteración y (max p: $0 \leq p \leq i+1$: S(p,i+1))

Ejercicio: Hacer un programa que calcule el máximo entre el valor de una variable **suma** (inicializada en cero) y (max p: $0 \leq p \leq N$: S(p,N)), para $N \geq 0$.

Notemos de nuevo que:

(max p: $0 \leq p \leq i+1$: S(p,i+1))
= podemos separar el último término pues $i \geq 0$
max ((max p: $0 \leq p \leq i$: S(p,i+1)), S(i+1,i+1))
= por definición de S(p,q)
max ((max p: $0 \leq p \leq i$: S(p,i+1)), 0)
= por definición de S(p,q) y $i \geq 0$
max ((max p: $0 \leq p \leq i$: S(p,i)+A[i]), 0)
= por aritmética y $i \geq 0$ (+ se distribuye sobre max)
max ((max p: $0 \leq p \leq i$: S(p,i)) + A[i], 0)

Así:

(max p: $0 \leq p \leq i+1$: S(p,i+1)) = max ((max p: $0 \leq p \leq i$: S(p,i)) + A[i], 0)

Por lo tanto si al comienzo de la iteración i tenemos en una variable **r** la cantidad (max p: $0 \leq p \leq i$: S(p,i)) entonces al comienzo de la iteración i+1 podemos tener la misma cantidad para i+1, si en *Prog* hacemos la asignación **r := (r + A[i]) max 0**, donde **a max b** representa en máximo entre a y b. Luego, como ya vimos, este valor de r nos sirve para calcular el valor de **suma** al comienzo de la iteración i+1, haciendo la asignación **suma := suma max r**. Y podemos agregar r = (max p: $0 \leq p \leq i$: S(p,i)) a nuestro invariante (***aplicando la técnica de fortalecimiento del invariante***). Finalmente en *Prog* debemos incrementar i en 1 para reflejar la siguiente iteración.

El programa finalmente es:

[const N: entero;
const A: arreglo [0..N) de entero;

```
  var suma: entero;
  { N ≥ 0 }
  suma := 0;
  i := 0;
  { Invariante: (suma = (max p,q: 0 ≤ p ≤ q ≤ i : S(p,q))) ∧ 0 ≤ i ≤ N
                ∧ r = (max p: 0≤p≤i: S(p,i)) }
  do  i<N  →  r := (r + A[i]) max 0;
              suma := suma max r;
              i := i+1
  od
  { suma = (max p,q: 0 ≤ p ≤ q ≤ N : (∑ j: p ≤ j < q : A[j]))  }
]
```

Note lo simple de la solución si hacemos un buen análisis. La estrategia de solución que empleamos se denomina Divide-and-Conquer (divide y conquistarás), y consiste en expresar la solución del problema en términos de la solución de problemas "más pequeños" del mismo tipo que el original. En el ejemplo anterior teníamos:

$$(\max p,q: 0 \le p \le q \le N : S(p,q)) = \max ((\max p,q: 0 \le p \le q \le N-1 : S(p,q)), (\max p: 0 \le p \le N : S(p,N)))$$

Más adelante seguiremos ejercitando esta estrategia para conseguir soluciones a problemas.

Ver la derivación del mismo programa en Kaldewaij (pag.67-70).

Ejercicios:

1) Páginas 62 y 71 del Kaldewaij.
2) Hacer un programa que satisfaga la siguiente especificación:

```
[  const N: entero;
   const s: secuencia de enteros;
   var r: entero;
   { N ≥ 0 }
   S
   { r = (#i,j: 0 ≤ i < j < N: s[i] ≤0 ∧ s[j] ≥ 0) }
]
```

Ejemplo: Dada una matriz cuadrada de orden N de enteros hacer un programa que determine si la matriz es simétrica.

La especificación formal sería:

```
[  const N: entero;
   const M: arreglo [0..N)×[0..N) de entero;
   var simetrica: booleano;
```

```
  { N ≥ 0 }
  es simétrica
  { simetrica ≡ (∀i,j: 0 ≤ i, j < N: M[i][j] = M[j][i]  }
]
```

El programa sería:

```
[  const N: entero;
   const M: arreglo [0..N)×[0..N) de entero;
   var simetrica: booleano;
   var n, m: entero;
   { N ≥ 0 }
   simetrica := verdad;
   n := 0;
   do n ≠ N ∧ simetrica →
         m := n;
         do  (m ≠ N) ∧ simetrica →
             if M[n][m] ≠ M[m][n] → simetrica := falso
             [] M[n][m] = M[m][n] → skip
             if;
             m := m+1
         od;
         n := n+1
   od
   { simetrica ≡ (∀i,j: 0 ≤ i, j < N: M[i][j] = M[j][i]  }
]
```

Note que el programa anterior se deduce del invariante para el primer ciclo siguiente:

$$\text{simetrica} \equiv (\forall i: 0 \le i < n : (\forall j: 0 \le j < N: M[i][j] = M[j][i])) \ \wedge \ 0 \le n \le N$$

Ejercicio: probar la correctitud del programa anterior.

6.2. Manipulación de arreglos

En esta sección presentamos cómo razonar sobre arreglos en programas que puedan modificarlos.

Dado un arreglo b, es decir, una función con dominio un segmento [p..q), es posible en el pseudolenguaje modificar el valor de la variable b[i], para un i dado, mediante una asignación. Así, b[i]:=e es una instrucción válida del pseudolenguaje, donde i es una expresión cuyo valor debe estar en el segmento [p..q), y e es una expresión con igual tipo que los elementos del arreglo b. La interpretación operacional es “reemplazar el valor de la variable b[i] por el valor resultante de evaluar e”.

Veremos que esta asignación se diferencia de la asignación ordinaria, en el hecho de que

esta afecta a la función b y no sólo a la variable b[i], y si no estamos conscientes de esto podemos llegar a conclusiones incorrectas. Veamos un ejemplo:

Suponga que b[0] = 1 y b[1] = 1. Entonces, b[b[1]] = b[1] = 1 y la instrucción b[b[1]] := 0 es equivalente a b[1] := 0 y después de esta asignación tendremos b[b[1]] = b[0] = 1. Parece paradójico que habiendo asignado 0 a b[b[1]] al final contenga 1; sin embargo, veremos que esto se debe a que estamos afectando el valor de una función completa, y no sólo el valor de una variable simple. Concluimos entonces que se cumple:

$$\{ b[0] = 1 \wedge b[1] = 1 \} \; b\,[b[1]] := 0 \; \{ b[b[1]] = 1 \}$$

Vemos la diferencia con la asignación a una variable x, donde se cumple:

$$\{ \text{verdad} \} \; x := 0 \; \{ x = 0 \}$$

Para expresar el cambio de valor de un arreglo (por lo tanto, de una función) introducimos la siguiente notación:

Sea **b** un arreglo, i una expresión cuyo valor está en el dominio de **b**, y e una expresión del tipo de los elementos del arreglo. Entonces b(i:e) denotará el arreglo (la función) que es igual a b salvo que la imagen de i es e:

$$b(i:e)[j] = \begin{cases} e & \text{si} \quad i = j \\ b[j] & \text{si} \quad i \neq j \end{cases}$$

Por ejemplo, si b=<2, 4, 6> con dominio [0..3) entonces:

- b(0:8)[0] = 8 (es decir, la función b(0:8) aplicada a 0 da 8)
- b(0:8)[1] = b[1] = 4 (es decir, la función b(0:8) aplicada a 1 da 4)
- b(0:8)[2] = b[2] = 6 (es decir, la función b(0:8) aplicada a 2 da 6)
- b(1:8) = <2, 8, 6> con dominio [0..3)
- (((b(0:8))(2:9)) = <8, 4, 9> con dominio [0..3)
- (((b(0:8))(0:9)) = <9, 4, 6>, con dominio [0..3), para simplificar paréntesis colocaremos b(0:8)(0:9).

Será necesario saber simplificar expresiones con la nueva notación. Por ejemplo, queremos simplificar b(i:5)[j] = 5, es decir, tratar de encontrar un predicado equivalente pero en términos de b solamente. Podemos seguir el siguiente razonamiento: un j dado cumple con $j=i \vee j \neq i$. Para j=i tenemos que b(i:5)[j] = 5 se reduce a b(i:5)[i] = 5 y esto a 5=5; para $j \neq i$, se reduce a b[j]=5. Así:

b(i:5)[j] = 5
≡ por tercero excluido y neutro de $\wedge$
$(i=j \vee i \neq j) \wedge b(i:5)[j] = 5$
≡ distributividad de $\wedge$ sobre $\vee$
$(i=j \wedge b(i:5)[j] = 5) \vee (i \neq j \wedge b(i:5)[j] = 5)$

≡ por definición de b(i:5)
(i=j ∧ 5=5) ∨ (i≠j ∧ b[j] = 5)
≡ 5=5 es verdad y simplificación del ∧
(i=j) ∨ (i≠j ∧ b[j] = 5)
≡ distributividad de ∨ respecto a ∧
(i=j ∨ i≠j) ∧ (i=j ∨ b[j] = 5)
≡ ley del tercero excluido
V ∧ (i=j ∨ b[j] = 5)
≡ simplificación del ∧
i=j ∨ b[j] = 5

Ejercicios: página 92 del Gries.

Sea b un arreglo de dimensión 2 con dominio [0..2)×[1..4) en los enteros, b=<<-4, -5, 2>, <5, 0, -3>>. Note que la notación b(x:A), donde x es una expresión definida en el segmento [0..2) representa a un arreglo de dimensión 2 con dominio [0..2)×[1..4) y cuyos elementos son los mismos de b salvo en x donde su valor es A, note que A debe ser un arreglo de enteros con dominio [1..4). Extendemos esta notación como sigue: b([x][y]:A) representa un arreglo de dimensión 2 con dominio [0..2)×[1..4) y cuyos elementos son todos iguales a los de b salvo el elemento [x][y] cuyo valor es A. Note que b([1][2]:10)[1] = b[1](2:10)=<5,10,-3>.

Ahora estamos en condiciones de explicar la asignación b[i] := e, en términos de la definición como función de los arreglos. La asignación b[i] := e no es más que una abreviación de la siguiente asignación a la variable b:

b := b(i:e)

Cuando describimos b[i] := e como una abreviación de b := b(i:e), la definición como función de b es usada para describir el *efecto* de la ejecución de la asignación, pero no para indicar cómo se implementa la asignación. En la ejecución de la asignación b[i]:=e vemos al arreglo como una colección de variables independientes: se evalúa i y e, se selecciona la variable b[i] cuyo índice es el valor de i, y se asigna el valor de e a b[i]. No se crea un nuevo arreglo completo b(i:e) que se asigna a b. Esto lo vemos mejor en la figura 8.

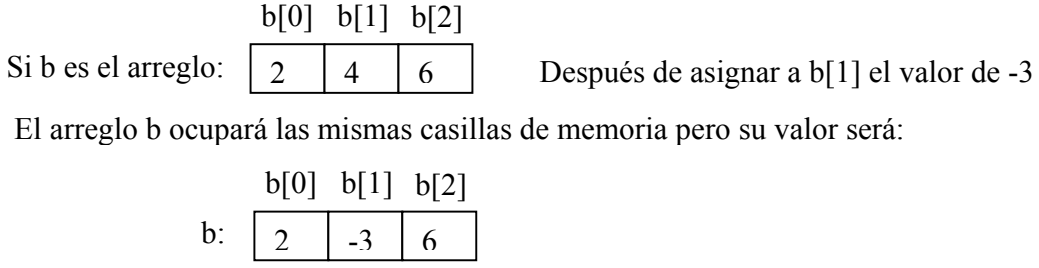


Figura 8

Por lo tanto la definición formal de la asignación a un elemento de un arreglo b con dominio [p..q), **la regla de la asignación a arreglos**, es:

{ P } b[i] := e {Q} se cumple
si y sólo si
[P $\Rightarrow$ e está bien definida $\wedge$ i está bien definida $\wedge$ p $\leq$ i < q $\wedge$ Q(b := b(i:e))]

donde Q(b := b(i:e)) denota a Q reemplazando las ocurrencias de b por b(i:e).

Ahora estamos en capacidad de probar que se cumple:

{ b[0] = 1 $\wedge$ b[1] = 1 } b [b[1]] := 0 { b[b[1]] = 1 }

Supongamos que se cumple b[0] = 1 $\wedge$ b[1] = 1, entonces:

b[b[1]] (b := b(b[1]:0))
= { sustitución }
b(b[1]:0)[b(b[1]:0)[1]]
= { b[1] = 1 }
b(1:0)[b(1:0)[1]]
= { definición de b(x:A) y 1 = 1 }
b(1:0)[0]
= { definición de b(X:A) y 1≠0 }
b[0]
= { b[0] = 1 }
1

Ejemplo donde se usa asignación de valores a elementos de un arreglo:

Problema: colocar en cero todos los elementos de un arreglo h de largo N≥0.

La especificación formal sería:

```
[  const N: entero;
   var h: arreglo [0..N) de entero;
   todos cero
   { (∀i: 0 ≤ i <N: h[i] = 0) }
]
```

Reemplazando la constante N por una variable entera n nos lleva a los invariantes P_0 y P_1 siguientes:

P_0: $(\forall i: 0 \leq i < n: h[i] = 0)$
P_1: $0 \leq n \leq N$

La guardia sería n ≠ N y los invariantes son establecidos inicialmente con n := 0. Investigamos ahora un incremento de 1 en n suponiendo que se cumple $P_0 \wedge P_1 \wedge n \neq N$:

$(\forall i: 0 \leq i < n+1: h[i] = 0)$
≡ como $0 \leq n$, podemos separar la sumatoria
$(\forall i: 0 \leq i < n: h[i] = 0) \wedge h[n] = 0$
≡ por definición de h(x:A)
$(\forall i: 0 \leq i < n: h[i] = h(n:0)[i]) \wedge h[n]=h(n:0)[n]$
≡ compactando la sumatoria
$(\forall i: 0 \leq i < n+1: h[i] = h(n:0)[i])$

La última línea nos dice que si reemplazamos a h por h(n:0), es decir, si asignamos 0 a h[n], se cumple $P_0(n := n+1)$. Note que $P_1(n := n+1)$ se cumple directamente.

Esto nos lleva a la siguiente solución de *todos cero*:

```
[  const N: entero;
   var h: arreglo [0..N) de entero;
   var n: entero;
   n := 0;
  ;do  n ≠ N → h[n] , n := 0, n+1 od
    { (∀i: 0 ≤ i <N: h[i] = 0) }
]
```

Ejemplo:
Calcular una tabla de frecuencias para N lanzamientos de un dado.

La especificación formal sería:

```
[  const N: entero; X: arreglo [0,N) de entero;
   var  h: arreglo [1..7) de entero;
  {  N ≥ 0  ∧ (∀i: 0 ≤ i < N: 1≤X[i] ≤ 6 )
   tabla de frecuencias
    { (∀i: 1 ≤ i < 7: h[i] = (#k: 0 ≤ k < N: X[k] = i))  }
]
```

Reemplazando la constante N por la variable n nos lleva a los invariantes:

P_0: $(\forall i: 1 \leq i < 7: h[i] = (\#k: 0 \leq k < n: X[k] = i))$
P_1: $0 \leq n \leq N$

La guardia sería n ≠ N y los invariantes son establecidos inicialmente con n := 0 y estableciendo luego la condición $(\forall i: 1 \leq i < 7: h[i] = 0)$, que ya hemos visto en el ejemplo anterior inicializar un arreglo en cero.

Investigamos ahora un incremento de 1 en n suponiendo que se cumple $P_0 \wedge P_1 \wedge n \neq N$. Para cualquier i, $1 \leq i \leq 6$, tenemos:

$(\#k: 0 \leq k < n+1: X[k] = i)$

= como $0 \leq n$, podemos separar la sumatoria

$(\#k: 0 \leq k < n: X[k] = i) + \#(X[n]=i)$

= análisis de casos

$$\begin{cases} (\#k : 0 \leq k < n : X[k] = i) & si \;\; i \neq X[n] \\ (\#k : 0 \leq k < n : X[k] = i) + 1 & si \;\; i = X[n] \end{cases}$$

= por P_0

$$\begin{cases} h[i] & si \; i \neq X[n] \\ h[X[n]] + 1 & si \; i = X[n] \end{cases}$$

= por definición de h(x:A)

$h(X[n]:h[X[n]]+1)[i]$

Por lo tanto, h debe ser reemplazado por h(X[n]:h[X[n]]+1). Y llegamos a la siguiente solución de *tabla de frecuencias*:

```
[  const N: entero; X: arreglo [0..N) de entero;
   var  h: arreglo [1..7) de entero;
   var n: entero;
  {  N ≥ 0  ∧ (∀i: 0 ≤ i < N: h[i] ≤ 6 )
   n := 0;
   [ var m: entero; m := 1; do  m ≠ 7 → h[m] , m := 0, m+1  od]
  ;do  n ≠ N →
                  h[X[n]] , n := h[X[n]]+1, n+1
   od
    { (∀i: 1 ≤ i < 7: h[i] = (#k: 0 ≤ k < N: X[k] = i))  }
]
```

En el programa anterior hemos introducido un nuevo constructor que llamaremos “bloque” y que corresponde al programa completo:

```
[ var m: entero; m := 1; do  m ≠ 7 → h[m] , m := 0, m+1  od]
```

En un bloque, el alcance de las variables que se declaran en él es el bloque mismo. Por ejemplo, la variable m no se puede utilizar fuera del bloque. Sin embargo toda variable declarada en un bloque B puede ser utilizada en los bloques que se definan dentro de B; por ejemplo, el arreglo h (declarado en el bloque del programa principal) puede ser utilizado dentro del bloque donde se declara la variable m.

Podemos tener anidamiento de bloques:

```
[var n:... [var m:... [ var p:...]  ...]  ...]
```

El identificador de una variable declarada en un bloque dado B debe ser distinto de los identificadores de las variables declaradas en los bloques que contienen a B.

Ejercicio:

1) Demuestre que si h es un arreglo de enteros con dominio [0,N), P es el predicado "$0 \le n \le N \wedge (\forall i: 0 \le i < n: h[i] = H(i))$" y H(i) es una expresión en la cual no aparece h[], entonces se cumple:

$$\{ P \wedge n \neq N \wedge E = H(n) \}\ h[n] := E\ \{ P(n := n+1) \}$$

(Note que asignar ceros en un arreglo es un caso particular de esta proposición cuando H(i)=0).

6.3. Intercambio de dos elementos de un arreglo

Muchos problemas de programación pueden ser resueltos intercambiando los elementos de un arreglo. Denotaremos por **intercambio(h,E,F)** a la acción:: "intercambiar los valores de h[E] y h[F]". Esta interpretación informal no ayuda mucho. Una definición formal la daremos definiendo el arreglo h(x, y : A, B) de la siguiente forma:

$$h(x, y : A, B)[i] = \begin{cases} h[i] & \text{if } i \neq x \wedge i \neq y \\ A & \text{if } i = x \\ B & \text{if } i = y \end{cases}$$

Por lo tanto **la regla de intercambio de elementos de un arreglo** es la siguiente:

{ P } intercambio(h,E,F) { Q } se cumple si y sólo si

[P $\Rightarrow$ E está bien definida $\wedge$ F está bien definida $\wedge$ Q(h := h(E, F : h[F], h[E]))] es una tautología.

Si E y F no dependen de h, es fácil predecir el efecto de intercambio(h,E,F) sin hacer una demostración formal. De lo contrario, es difícil predecir el efecto sin hacer los cálculos correspondientes. Por ejemplo, sea h[0]=0 y h[1]=1. Por lo tanto intercambio(h,h[0],h[1]) es lo mismo que intercambio(h,0,1) lo cual resulta en h[0]=1 y h[1]=0. En particular, tendremos h[h[1]]=h[0]=1. Por lo que:

{ h[h[0]]=0 } intercambio(h,h[0],h[1]) { h[h[1]] = 0 } NO SE CUMPLE

Si E y F no dependen de h entonces la operación intercambio(h,E,F) puede escribirse como:

[var r: entero; r := h[E]; h[E] := h[F]; h[F]:= r]

Existe una regla, llamada **la regla de intercambio simple**, que es muy útil cuando queremos probar la correctitud de programas que involucran intercambio de elementos de un arreglo:

Si h no aparece en las expresiones E y F entonces se cumple:

{ $(\forall i: i \neq E \wedge i \neq F : h[i] = H(i)) \wedge h[E]=A \wedge h[F]=B$ }
intercambio(h,E,F)
{ $(\forall i: i \neq E \wedge i \neq F : h[i] = H(i)) \wedge h[E]=B \wedge h[F]=A$ }

Ejercicios:

1) Demuestre formalmente que se cumple:

$$\{ h[0]=0 \wedge h[1]=1 \} \text{ intercambio}(h,h[0],h[1]) \{ h[h[1]] = 1 \}$$

2) Demuestre formalmente la regla de intercambio simple.

Por lo tanto esta regla la podemos utilizar para demostrar la correctitud de programas que realicen intercambios.

Ejemplo: (reubicación respecto a un elemento pivote) Dado un arreglo h de N enteros queremos reubicar los elementos del arreglo utilizando sólo intercambio de elementos como la única operación permitida de arreglos (aparte de la operación de observación de sus elementos), de forma tal que los primeros k elementos del arreglo, hasta un cierto k a determinar, sean menores o iguales que un número entero dado X, y los últimos N-k elementos del arreglo desde k sean mayores que X.

La especificación formal de este problema es:

[const N, X: entero;
  var h: arreglo [0..N) de entero;
  var k: entero;
  { $N > 0 \wedge h = H$ }
  redistribuir
  { $(\forall z: z \in Z : (\#j: 0 \leq j < N : h[j] = z) = (\#j: 0 \leq j < N : H[j] = z)) \wedge 0 \leq k \leq N$
    $\wedge (\forall i: 0 \leq i < k : h[i] \leq X) \wedge (\forall i: k \leq i < N : h[i] > X)$ }
]

El predicado $(\forall z: z \in Z : (\#j: 0 \leq j < N : h[j] = z) = (\#j: 0 \leq j < N : H[j] = z))$ indica que el valor final de h es una permutación de su valor original (están los mismos elementos originales y en igual cantidad). Los otros dos predicados establecen la propiedad que cumplirá h una vez concluya el programa.

Una forma de atacar el problema es utilizando la técnica de reemplazo de constante por variable, por ejemplo N por n en el último predicado. Sin embargo, esto resulta en un programa complicado pues no aprovecha la simetría del problema que indica que los primeros elementos del arreglo son menores o iguales que X y los últimos mayores que X (la simetría de problema sugiere que a medida que vayamos considerando uno a uno los

elementos originales del arreglo, ir colocando los menores o iguales al comienzo y los mayores al final del mismo arreglo).

Note que si para p,q, $0 \leq p \leq q \leq N$, tenemos que los primeros p elementos de h, el segmento h[0,p), son menores o iguales que X y los últimos N-q elementos de h, el segmento h[q,N), son mayores que X, entonces, si $p \neq q$, podemos decidir fácilmente donde reubicar al elemento h[p] o al elemento h[q-1] (los extremos del segmento h[p,q)) de forma que aumentemos en 1 el número de elementos ya ubicados sea al comienzo o al final del arreglo dependiendo respectivamente de si es menor o igual, o mayor que X . Si h[p] es menor o igual que X, no hace falta reubicar a h[p] y tendremos que los primeros p+1 elementos de h son menores o iguales a X y los últimos N-q elementos de h son mayores que X. Si h[p] es mayor que X entonces podemos intercambiar los elementos h[q-1] y h[p], y así obtendremos que los primeros p elementos de h son menores o iguales a X y los últimos N-q+1 elementos de h son mayores que X. De esta forma, continuando este proceso llegamos a reubicar todos los elementos de h y el valor final de p será el k buscado en nuestro problema original.

Esto sugiere que el invariante sea:

$(\forall z: z \in Z : (\#j: 0 \leq j < N : h[j] = z) = (\#j: 0 \leq j < N : H[j] = z)) \wedge (\forall i: 0 \leq i < p : h[i] \leq X) \wedge (\forall i: q \leq i < N : h[i] > X) \wedge 0 \leq p \leq q \leq N$

Note que reemplazamos en la postcondición una misma variable (k) por dos distintas (p y q). Sin embargo hemos podido dejar igual la primera ocurrencia de k y reemplazar la segunda ocurrencia de k por la variable nueva q, quedando el invariante:

$$P_0 \wedge P_1 \wedge P_2 \wedge P_3$$

Donde:
P_0: $(\forall z: z \in Z : (\#j: 0 \leq j < N : h[j] = z) = (\#j: 0 \leq j < N : H[j] = z))$
P_1: $(\forall i: 0 \leq i < k : h[i] \leq X)$
P_2: $(\forall i: q \leq i < N : h[i] > X)$
P_3: $0 \leq k \leq q \leq N$

La postcondición se obtiene con k = q, por lo que la guardia será $k \neq q$. El invariante puede ser establecido inicialmente con k, q := 0, N. Como k debe aumentar o q disminuir en cada iteración, una función de cota decreciente sería q-k la cual es no negativa pues q es siempre mayor o igual a k.

Nuestro programa tiene el esquema siguiente:

k, q := 0, N;
do $k \neq q \rightarrow$ S od

Determinemos S. Si $k \neq q$ entonces k < q, y los elementos del arreglo en el segmento [k,q) son todos candidatos a ser reubicados, los más obvios a considerar son h[k] o h[q-1].

Consideremos h[k], lo cual nos lleva al esquema:

k, q := 0, N;
do k ≠ q
→ if h[k] ≤ X → S_1
[] h[k] > X → S_2
fi

od

Como ya observamos, si h[k] ≤ X entonces basta con incrementar k en 1 para que se siga cumpliendo el invariante, y si h[k] > X entonces basta con intercambiar h[p] con h[q-1] (al ser k < q, tenemos k ≤ q-1) y disminuir q en 1 para que se siga cumpliendo el invariante. El programa final con las anotaciones es el siguiente:

[const N, X: entero;
var h: arreglo [0..N) de entero;
var k,q: entero;
{ N > 0 ∧ h = H }
k, q := 0, N;
{Invariante: $P_0 \wedge P_1 \wedge P_2 \wedge P_3$, demo 0; función de cota decreciente: q-k}
do k ≠ q
→ { $P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge k \neq q$ }
if h[k] ≤ X → k := k+1
[] h[k] > X → intercambio(h,k,q-1); q := q-1
fi
{ $P_0 \wedge P_1 \wedge P_2 \wedge P_3$, demo 1}
od
{ (∀z: z ∈ Z : (#j: 0 ≤ j < N : h[j] = z) = (#j: 0 ≤ j < N : H[j] = z)) ∧ 0 ≤ k ≤ N
∧ (∀i: 0 ≤ i < k : h[i] ≤ X) ∧ (∀i: k ≤ i < N : h[i] > X), demo 2; Terminación: demo 3}
]

Ejercicios:

1) Demuestre la correctitud del programa anterior (ayuda: utilice la regla de intercambio simple).
2) Utilizando intercambios como única operación sobre arreglos, hacer un programa que dado un arreglo de enteros de largo N, reubique los elementos del arreglo de forma tal que los primeros elementos del arreglo sean menores que un pivote dado X, los elementos del medio del arreglo sean iguales a X y los últimos elementos del arreglo sean mayores que X..
3) Utilizando intercambios como única operación sobre arreglos, hacer un programa (rotación de los elementos de un arreglo) que cumpla:
[const K, N: entero;
var h: arreglo [0,N) de entero;
{ N ≥0 ∧ h = H }
rotación

```
    { (∀i: 0 ≤ i < N : h[(i+K) mod N]] = H[i] )
[
```

Cuando definimos una función o un procedimiento, la manera como declaramos un parámetro formal tipo arreglo en el pseudolenguaje es como sigue:

proc <nombre procedimiento> (... ; <tipo parámetro> x : arreglo de <tipo>; ....)

donde,

- <nombre procedimiento> es el nombre del procedimiento.
- <tipo parámetro> es entrada, salida o entrada-salida. La interpretación operacional es exactamente la misma que para variables simples, es decir, si el parámetro formal es de entrada entonces en una llamada al procedimiento se copia el valor del parámetro real en la variable correspondiente al parámetros formal, por lo tanto en tiempo de ejecución el parámetro formal es un arreglo que ocupa un lugar de memoria completamente distinto al del parámetro real.

En una llamada a un procedimiento que contenga arreglos como parámetros, los parámetros reales determinan los límites inferiores y superiores del los rangos de los parámetros formales. Por ejemplo, si en la llamada el parámetro real es un arreglo de enteros con dominio [4,10) entonces en la ejecución de la llamada el parámetro formal será un arreglo de enteros con dominio [4,10). Por lo tanto es conveniente pasar también como parámetros a los límites inferior y superior de los dominios de cada arreglo que se pase como parámetro.

Por ejemplo, el programa de reubicación según un pivote visto antes lo podemos convertir en un procedimiento que dado un arreglo y un pivote, redistribuya los elementos del arreglo según el pivote:

```
{ Pre: h = H ∧ H es un arreglo con dominio [N1..N2) ∧ 0 ≤ N1 ≤ N2 }
{ Post: (∀z: z ∈ Z : (#j: N1 ≤ j < N2 : h[j] = z ) = (#j: N1 ≤ j < N2 : H[j] = z ))
        ∧ N1 ≤ k ≤ N2 ∧ (∀i: N1 ≤ i < k : h[i] ≤ X ) ∧ (∀i: k ≤ i < N2 : h[i] > X ) }
proc Pivote( entrada X : entero; entrada N1, N2: entero; entrada-salida h: arreglo de entero;
             salida k: entero)
[
  var q: entero;
   k, q := N1, N2;
  do k ≠ q
       → if h[k] ≤ X → k := k+1
          [] h[k] > X → intercambio(h,p,q-1); q := q-1
          fi
  od
]
```

Ejercicios:

1) Hacer un procedimiento que implemente la operación de intercambio de dos elementos de un arreglo dado.
2) Hacer una función que reciba como parámetro una matriz cuadrada y devuelva verdad si y sólo si la matriz es simétrica.
3) Página 168 Kaldewaij

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.

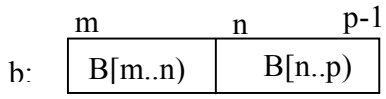
En un lenguaje de programación moderno, como JAVA, que permite *definir clases de objetos,* el refinamiento de datos se efectúa construyendo una unidad de programa aparte, llamada igualmente *clase*, que contiene segmentos de programa que implementan por una parte la representación de los objetos en 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

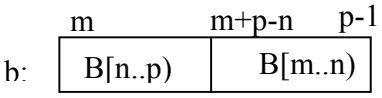
El término "más simple" puede significar diferentes cosas según el contexto. Un problema puede ser más simple debido a que algunas de sus restricciones han sido omitidas (resultando en una generalización del problema). El problema puede ser más simple porque, al contrario, le hemos agregado restricciones. 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:

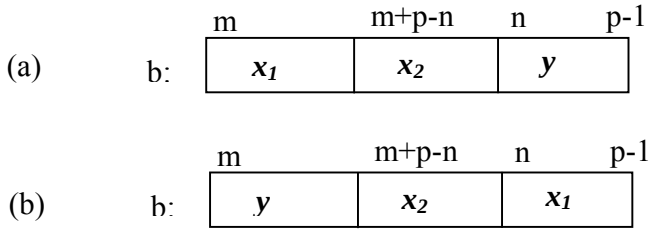


El programa sólo podrá declarar un número constante de variables adicionales de tipos básicos y utilizar sólo operaciones de intercambio de dos elementos de un arreglo más las propias a de los tipos básicos.

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.

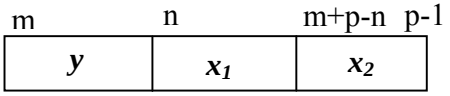
Suponga por el momento que el segmento b[m..n) es mas largo que b[n..p). Consideremos que el segmento b[m..n) consiste de dos segmentos, el primero de los cuales es del mismo largo que b[n..p) (ver diagrama (a) dado más abajo). Entonces los segmentos de igual largo x_1 y y pueden ser intercambiados obteniéndose 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.



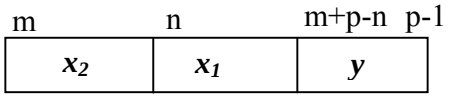
Ahora supongamos que en lugar del primer segmento del problema original, es el segundo segmento el que tiene largo mayor, es decir, b[n..p) es más grande. Este caso es ilustrado en

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

(c) b:



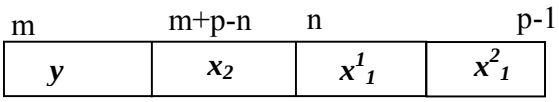
(d) b:



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.

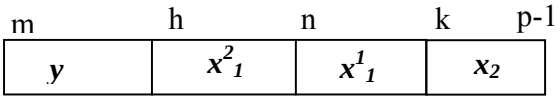
Supongamos que en (b) x_1 es mas largo que x_2, tendremos la situación siguiente:

(b') b:



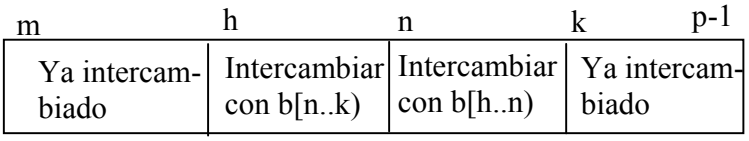
Donde $x_1 = x^1_1 \,||\, x^2_1$, x_2 y x^2_1 tienen el mismo largo. Después de intercambiar x_2 y x^2_1 mediante *IntercambiarIguales* obtenemos:

(b'') b:



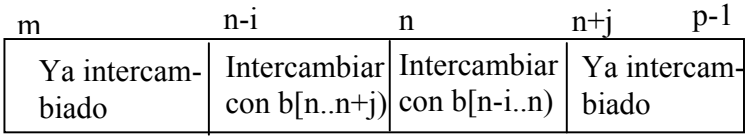
Note que los segmentos [m..h) y [k..p) ya están en el sitio que les corresponde, y hay que intercambiar los subsegmentos [h..n) y [n..k) del segmento [h..k). Por lo tanto, podemos obtener el invariante:

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  ∧   (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 )   }
]
```

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 + √disc) / 2*A;
                                        x2 := (-B – √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  ∧  B²- 4.A.C = 0 ) )
                              ∧ (x1 = x1a[0]+x1a[1].i)  ∧   ( A.x1² + B.x1 + C = 0 ) )      ∨
          ( n = 2  ∧  A ≠ 0  ∧  ( A.x1² + B.x1 + C = 0 ) ∧ (B² – 4AC ≠ 0)
                              ∧ (x1 = x1a[0] + x1a[1].i) ∧ (x2 = x2a[0] + x2a[1].i)
                              ∧   ( A.x2² + 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 ∧ 1 ≤*nuevoi* ≤5 ∧ *carnets* = *C* ∧ *indices* = *I* ∧ /*carnets* /< *MAX* ∧
nuevoc ∉ *carnets* }
{Post: *carnets* = *C* ∪{*nuevoc*} ∧ *indices* = *I* ∪{(*nuevoc*, *nuevoi*)}}

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

metodo *UnMejorEstudiante*(**salida** *c*: **entero**; **salida** *i*: **real**)
no modifica: *carnets, indices*
{Pre: *carnets* ≠ ∅}
{Post: *c* ∈ *carnets* ∧ *i* = *indices (c)* ∧ (∀ *otroc*: *otroc* ∈ *carnets*: *indices*(*otroc*)≤ *i*) }

booleano metodo *HayEstudiantes*( )
no modifica: *carnets, indices*
{Pre: verdad }
{Post: *Devuelve el valor de la expresión (carnets* ≠ ∅) }

booleano metodo *EstaEstudiante*(entrada c: entero)
no modifica: *carnets, indices, MAX*
{Pre: verdad }
{Post: Devuelve *(c* ∈ *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 ≤ *cantidad* ≤ MAX ∧
(∀i: 0 ≤ i < *cantidad*: *seccarnets*[i] > 0 ∧ 1 ≤ *secindices*[i] ≤ 5) ∧
(∀i: 0 ≤ i < *cantidad* ∧ 0 ≤ j < *cantidad*:
i ≠ j ⇒ *seccarnets*[i] ≠*seccarnets*[j]) }

{**Invariante de Acoplamiento**: carnets = { *seccarnets*[i] : 0 ≤ i < *cantidad* }∧

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
```

od
{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 $A.i < |A.S|$

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: (∃S1,S2: (S = S1 || S2) ∧ (esta = existe un elemento en S1 que cumple P(x)) ∧ (i =
|S1|) );
  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.

8. Soluciones recursivas de problemas

8.1. Planteamiento de soluciones recursivas de problemas

En este capítulo hablaremos de soluciones recursivas de problemas, es decir, soluciones que vienen expresadas en función de soluciones a problemas mas sencillos del mismo tipo que el problema original. Esta estrategia de solución de problemas se conoce como *divide y conquistarás*.

Se trata de resolver un problema mediante la solución de varios problemas similares (o del mismo tipo) mas pequeños. Por pequeño entendemos que el problema posee menor cantidad datos. Si la división puede ser llevada a cabo, entonces el mismo proceso puede ser aplicado a los problemas más pequeños.

Frecuentemente la estrategia "divide y conquistarás" consiste en dividir el problema original en dos o más problemas similares. Sin embargo, el mismo principio aplica cuando reducimos el problema original a sólo un problema similar más pequeño. Por ejemplo, para calcular el máximo elemento de un arreglo b de enteros de largo N, basta con calcular el máximo m_1 del segmento b[0..N-1) y y tomar como máximo m de b[0..N) al máximo entre m_1 y b[N-1]. Cuando b posee un solo elemento no es necesario, pues el máximo de b[0..1) es b[0]. La solución del problema queda de la forma:

"máximo de b[0..N) = max (máximo de b[0..N-1), b[N-1]) $\wedge$ máximo de b[0..1) = b[0]"

Este tipo de solución, en donde la solución del problema queda expresada en términos de la solución de un problema más pequeño del mismo tipo, la llamamos "solución recursiva" del problema.

Note que si el lenguaje de programación permitiese que un procedimiento o una función se llame a si mismo, entonces podríamos obtener un programa recursivo que resuelve nuestro problema. Los programas que haremos en este capítulo serán iterativos, pero partiendo de una primera versión recursiva de la solución del problema. Por ejemplo, ya hemos hecho un programa iterativo para el cálculo del n-ésimo número de Fibonacci, aún cuando la definición de éstos es recursiva: Fib(n)= Fib(n-1) + Fib(n-2), $n \geq 2$, Fib(0)=0, Fib(1)=1.

La diferencia entre la estrategia discutida en la sección 7.2.1 y la estrategia "divide y conquistarás" es muy sutil. En la primera estrategia, primero reconocemos un problema más simple y luego nos preguntamos cómo éste puede ser utilizado de manera efectiva. Este fue el caso del ejemplo que intercambiaba dos segmentos de un arreglo. En la estrategia "divide y conquistarás" primero nos preguntamos lo que significa dividir el problema en piezas más pequeñas y luego buscar la manera de resolver el problema original en términos de las piezas, y esto puede conllevar la solución de problemas más simples como la estrategia de la sección 7.2.1.

Veamos otros ejemplos donde se aplica la estrategia "divide y conquistarás".

Problema 1: Calculo del máximo de un arreglo de enteros.

Otra forma de formular la solución de este problema en términos de problemas mas pequeños del mismo tipo es:

Suponiendo que m-n > 1, el máximo de b[n..m) es el máximo de b[n..m-1) si b[n] ≥ b[m-1], si no es el máximo de b[n+1..m). Cuando m=n+1 el máximo es b[n].

Problema 2: Determinar la suma de los dígitos de un número natural n.

La solución se puede plantear como:

Suma de los dígitos de n = n mod 10 + (suma de los dígitos de (n div 10))

Problema 3: Cálculo de A^B, para A, B enteros y B entero no negativo.

$$A^B = \begin{cases} 1 & si \quad B = 0 \\ (A * A)^{B \; div \; 2} & si \quad B \bmod 2 = 0 \\ A * A^{B-1} & si \quad B \bmod 2 = 1 \end{cases}$$

De esta solución recursiva podemos deducir un programa iterativo eficiente para el cálculo de $r = A^B$ utilizando el invariante "$r*x^y = A^B \wedge 0 \leq y$". Más adelante explicaremos cómo obtener a partir de un esquema más general, el programa que a continuación se presenta:

```
[  r, x, y := 1, A, B;
   { Invariante: r*x^y = A^B ∧ 0≤y; cota: y }
   do  y ≠ 0
         → if    y mod 2 = 0  →  x, y := x*x, y div 2
             []    y mod 2 = 1  →  r,y := r*x, y-1
             fi
   od
]
```

Problema 4: (búsqueda binaria)

Dado un arreglo de enteros b[0..N) ordenado en forma creciente, determinar si un número x dado se encuentra en b.

Podemos utilizar el esquema de búsqueda secuencial visto en la sección 7.1. Sin embargo podemos aprovechar el acceso directo a los elementos de un arreglo y al hecho que éste está ordenado con el fin de conseguir un programa más eficiente.

La solución a este problema es similar a la que usamos cuando deseamos buscar un número de teléfono en una guía telefónica: no hacemos una lectura secuencial de la guía partiendo de la primera página hasta encontrar el número buscado (pasaríamos todo un día hasta

llegar al apellido Rodríquez). Lo que hacemos es abrir la guía en un lugar que nos parezca conveniente, y si el apellido de la persona está en las dos páginas donde abrimos, entonces paramos, si no determinamos si está mas allá de la página de la derecha o antes de la página de la izquierda dependiendo de si el apellido está antes o después.

La solución entonces es: comparar el elemento del medio b[N div 2] de b[0..N) con x, si son iguales entonces hemos encontrado el elemento en b, si no son iguales entonces se busca x en el segmento b[0..N div 2) o en el segmento b[N div 2 + 1 .. N) dependiendo respectivamente de si x es menor o mayor que b[N div 2] . Si el segmento es vacío, x no está en el segmento. La solución como vemos depende del largo del segmento, por lo que la podemos generalizar como sigue:

x está en b[n..m) si y sólo si ((m-n > 0) $\wedge$ (x = b[(m+n) div 2]
$\vee$ ((x < b [(m+n) div 2]) $\wedge$ (x está en b[n..(m+n) div 2)))
$\vee$ ((x > b [(m+n) div 2]) $\wedge$ (x está en b[(m+n) div 2 +1 .. m))))

Podemos entonces deducir el invariante siguiente (en forma parecida al ejemplo de intercambio de segmentos de las sección 7.1), reemplazando las constantes n y m por dos variables i y j:

P: x está en b[n..m) $\equiv$ x está en b[i,j) $\wedge$ $n \leq i \leq j \leq m$

inicialmente el invariante se establece con i, j := n, m y la guardia será i-j>1 pues cuando el arreglo b tiene al lo sumo un elemento es sencillo comprobar si el elemento x está en b. La función de cota será j-i y el programa sería:

```
i,j := n,m;
do j-i>1
      → k := (i+j) div 2;
         if   b[k] = x →  i :=k; j := k+1
         []   b[k] > x →  j := k
         []   b[k] < x →  i := k + 1
         fi
od;
if  i=j →  esta=falso
[]  i≠j →  esta= (x = b[i])
fi
{ esta ≡ x está en b[0..N)  ∧ (esta ⇒ x = b[i]) }
```

Problema 5: Torres de Hanoi.

8.2. Diseño iterativo de soluciones recursivas de problemas: Invariante de Cola.

En esta sección analizamos en un contexto general las soluciones iterativas de problemas que admiten una **solución recursiva de cola**. Por solución recursiva de cola entendemos una solución "divide y conquistarás" donde la solución del problema original depende de un solo problema más pequeño del mismo tipo, como los problemas 1 a 4 de la sección 8.1.

Supongamos que tenemos una función F que cumple:

$$F(x) = \begin{cases} h(x) & si\ b(x)\ es\ verdadero \\ F(g(x)) & si\ b(x)\ es\ falso \end{cases}$$

Vemos que el valor de la función en un punto x viene expresado en términos de la misma función para un punto g(x), por lo que este es un caso particular de una definición recursiva de cola de la función F. Queremos desarrollar un programa que calcule F(X) para X dado, es decir, que satisfaga la especificación siguiente:

```
[   const X: ...;
    var r: ...;
   { verdad }
    cálculo de F(X)
    {  r = F(X)  }
]
```

Si tomamos como invariante al predicado siguiente (llamado **invariante de cola**):

$$F(x) = F(X)$$

Obtenemos el siguiente programa iterativo:

```
[  var x;
   x := X;
   { Invariante:  F(x) = F(X)  }
   do  ¬ b(x)  →   x  :=  g(x)  od;
   r  :=  h(x)
]
```

Por lo tanto, resolver un problema por recursión de cola significa encontrar una función F con las características anteriores.

Ejemplos:

Problema 1: Cálculo del máximo de un arreglo de enteros con dominio [m.n) y n>m.

La especificación formal sería:

```
[  const N: entero;
   const b: arreglo [0..N) de entero;
   var r: entero;
   { N > 0 }
   máximo
   {  r = (max i : 0 ≤ i < N : b[i])  }
]
```

Una forma de formular la solución de este problema en términos de la solución de problemas mas pequeños del mismo tipo es:

Suponiendo que m-n > 1, el máximo de b[n..m) es el máximo de b[n..m-1) si b[n] ≥ b[m-1], si no es el máximo de b[n+1..m). Cuando m=n+1 el máximo es b[n]. En general, si definimos F(x,y) = (max i : x ≤ i < y : b[i]), con x < y, entonces tenemos que F tiene la siguiente definición recursiva de cola:

$$F(x,y) = \begin{cases} b[x] & si \quad y - x = 1 \\ F(x+1, y) & si \quad b[x] \le b[y-1] \wedge (y - x \neq 1) \\ F(x, y-1) & si \quad b[y-1] \le b[x] \wedge (y - x \neq 1) \end{cases}$$

Por lo tanto la especificación de la postcondición puede ser escrita como:

$$r = F(0,N)$$

Por lo tanto podemos proponer como invariante:

Inv: $F(x,y) = F(0,N) \wedge 0 \le x \le y \le N$

Y el programa *máximo* sería:

```
[ var x, y : entero
   x, y := 0, N;   {  N > 0  }
   { Inv: F(x,y) = F(0,N) ∧  0 ≤ x ≤ y ≤ N; cota: y-x }
   do y-x ≠1 →
         if b[x] ≤ b[y-1] → x := x+1
         [] b[y-1] ≤ b[x] → y := y-1
         fi
   od;
   { Inv ∧ y-x=1, por lo que se tiene b[x] = F(0,N)  }
   r := b[x]
]
```

Note que el programa anterior es un bloque interno del programa donde se definen las variables N y b; es decir, el programa completo sería:

```
[  const N: entero;
   const b: arreglo [0..N) de entero;
   var r: entero;
   { N > 0 }
    [ var x, y : entero
      x, y := 0, N;   {  N > 0  }
      { Inv: F(x,y) = F(0,N) ∧  0 ≤ x ≤ y ≤ N; cota: y-x }
      do y-x ≠1 →
         if b[x] ≤ b[y-1] → x := x+1
         [] b[y-1] ≤ b[x] → y := y-1
         fi
      od;
      { Inv ∧ y-x=1, por lo que se tiene b[x] = F(0,N)  }
      r := b[x]
    ]
   {  r = (max i : 0 ≤ i < N : b[i])  }
]
```

Problema 2: Hacer un programa que determine el máximo común divisor (MCD) de dos números enteros positivos A y B .

La especificación formal sería:

```
[  const A, B: entero;
   var r : entero;
   { A > 0 ∧  B > 0  }
   máximo común divisor
   {  r  =  MCD(A,B)  }
]
```

Por propiedades matemáticas se sabe que si x e y son enteros positivos:

$$MCD(x,y) = \begin{cases} x & si \quad x = y \\ MCD(x-y,y) & si \quad x > y \\ MCD(x,y-x) & si \quad y > x \end{cases}$$

Por lo tanto tenemos una definición recursiva de cola del Máximo Común Divisor, lo cual nos dice que podemos proponer como invariante a:

$$x > 0 \wedge \ y > 0 \ \ \wedge \ MCD(x,y) = MCD(A,B)$$

Y el programa sería:

```
[  var x,y: entero;
```

```
  x, y := A, B;
  do x > y → x := x-y
  [] y > x → y := y-x
  od;
  r := x
]
```

Un caso más general de recursión de cola viene dado cuando queremos calcular r = G(X), para X dado, donde la función G viene definida como sigue:

$$G(x) = \begin{cases} a \quad si \quad b(x) \\ h(x) \oplus G(g(x)) \quad si \quad \neg b(x) \end{cases}$$

donde ⊕ es una operación asociativa con elemento neutro e.

Este problema puede ser resuelto con un invariante de cola de la forma:

P: G(X) = r ⊕ G(x)

El cual puede ser interpretado como:

"el resultado" = "lo que ha sido calculado hasta el momento" ⊕ "lo que resta por calcular"

P puede ser establecido inicialmente con r, x := e, X. Además , si b(x) es verdad, entonces:

```
        G(X) = r ⊕ G(x)
≡       definición de G y B(x) verdadero
        G(X) = r ⊕ a
```

Y cuando ¬b(x) es verdadero:

```
        G(X) = r ⊕ G(x)
≡  definición de G y que se cumple ¬b(x)
        G(X) = r ⊕ (h(x) ⊕ G(g(x)))
≡  ⊕  es asociativa
        G(X) = (r ⊕ h(x) ) ⊕ G(g(x))
≡  definición de P
        P(  r, x := r ⊕  h(x), g(x)  )
```

Esto lleva al siguiente esquema de programa:

```
{  verdad  }
[  var x;
   x, r := X, e;
   { Invariante: G(X) = r ⊕ G(x)  }
```

```
  do ¬b(x) → x, r := g(x) , r ⊕ h(x) od
  { G(X) = r ⊕ a }
]
{ r = G(X) }
```

Ejemplo:

Problema 3: Dado un número natural X, hacer un programa que calcule la suma de los dígitos decimales de X. Por ejemplo, la suma de los dígitos de 4329 es 4+3+2+9=18.

Si G(X) representa la suma de los dígitos de X, se tiene que G puede ser definida de la siguiente forma:

$$G(x) = \begin{cases} 0 & si \quad x = 0 \\ x \bmod 10 + G(x div 10) & si \quad x > 0 \end{cases}$$

Y la especificación formal del programa sería:

```
[ const X: entero;
  var r: entero;
  { X ≥ 0 }
  suma dígitos
  { r = G(X) }
]
```

Aplicando los resultados anteriores, obtenemos como invariante:

$$G(X) = r + G(x) \ \wedge \ x \geq 0$$

Y obtenemos el programa *suma dígitos* es:

```
{ X ≥ 0 }
[ var x: entero;
  x, r := X, 0;
  { Invariante: G(X) = r + G(x) ∧ x ≥ 0; cota: x }
  do x ≠ 0 → x, r := x div 10, r + x mod 10 od
]
{ r = G(X) }
```

Problema 4: Calcular A^B con A, B enteros y $B \geq 0$.

Si G(x,y) es la función x^y, para $y \geq 0$, tenemos:

$$G(x,y)=\begin{cases}1 \quad si \quad y=0\\ 1*G((x*x),y \text{ div } 2) \quad si \quad y \bmod 2=0\\ x*G(x,y-1) \quad si \quad y \bmod 2=1\end{cases}$$

El invariante de cola correspondiente a G es:

$$r*x^y = A^B \ \wedge \ y \geq 0$$

Y el programa es el que ya vimos en la sección 7.2.2. problema 3:

```
{  B ≥ 0 }
[  var x, y: entero;
   r, x, y := 1, A, B;
   { Invariante: r*x^y = A^B ∧ 0≤y; cota: y }
   do  y ≠ 0
         → if    y mod 2 = 0  →  x, y := x*x, y div 2
             []    y mod 2 = 1  →  r,y := r*x, y-1
             fi
   od
]
{  r = A^B  }
```

Otros ejemplo: Quicksort pag. 226 de Gries.

Ejercicios:

1) Defina una función recursiva de cola para el problema de búsqueda binaria.
2) Diseñe una solución recursiva para ordenar en forma creciente los elementos de un arreglo de enteros.
3) Página 78 Kaldewaij

Bibliografía

- J. Castro, F. Cucker, X. Messeguer, A. Rubio, Ll. Solano, B. Valles. *Curso de Programación*. McGraw Hill. 1993.
- E. Dijkstra, W.H.J. Feijen. *A Method of Programming*. Addison Wesley. 1988.
- David Gries. *The Science of Programming*. Springer-Verlag, New York Inc. 1981.
- Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall. 1990.
- Oscar Meza. *Notas personales del curso Algoritmos y Estructuras de Datos I*.
- Alejandro Teruel. *Notas personales del curso Algoritmos y Estructuras de Datos I*.