

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 $Z \times Z \times Z$, 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 $x=A \wedge y=B$, el cual establece que justo antes de ejecutar el programa “máximo” el estado es $x=A \wedge y=B$, 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 precondition P y postcondición Q se denota de la siguiente manera:

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

y tiene la siguiente interpretación operacional:

Siempre que una ejecución de S comience en un estado que cumple la precondition 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 $\mathfrak{R} \times \mathfrak{R} \times \mathfrak{R} \times \mathfrak{R} \times \mathfrak{R}$ (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 ambigüedad 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 precondition "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 precondition ("verdad"). En términos de conjuntos podemos razonar como sigue para mostrar que todo estado del espacio de estados satisface la precondition "verdad": Dada la precondition $P(x)$, donde x es una variable sobre el espacio de estados, el conjunto I de estados que satisfacen $P(x)$ es $I = \{ x: x \text{ pertenece al espacio de estados} \wedge P(x) \}$. 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: $[(x \text{ pertenece al espacio de estados} \wedge V) \equiv x \text{ pertenece al espacio de estados}]$), 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 :

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

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 $\{ z \in B : z \text{ cumple } P(z) \}$, es equivalente a decir que A es subconjunto de B y $(\forall z: z \in B: (z \in A \equiv P(z)))$.

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 $(\forall z: z \text{ es natural: } (z \in C \equiv (z \text{ es primo}) \wedge (z \text{ divide a } X)))$ y el predicado $(\forall z: z \text{ es primo: } (z \in C \equiv (z \text{ divide a } X)))$? 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 $\forall z: z \text{ es primo: } ((z \text{ divide a } X) \equiv (z \in C))$ 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
 { (∃i: 0 ≤ i < |S| : S[i]=x) ∧ (∀i: 0 ≤ i < |S| : S[i]≤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 ≤ 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 precondition puede ser reforzada y la postcondition 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 $\{P\} S \{ \text{falso} \}$ 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 $(x=X \wedge y=Y \wedge X>10) \Rightarrow (x=X \wedge y=Y)$ es una tautología.

3) Regla de conjunción:

$\{P\} S \{Q\}$ se cumple y $\{P\} S \{R\}$ se cumple es equivalente a $\{P\} S \{Q \wedge R\}$ se cumple

4) Regla de la disyuntividad:

$\{P\} S \{Q\}$ se cumple y $\{R\} S \{Q\}$ se cumple es equivalente a $\{P \vee R\} S \{Q\}$ se cumple

Note que $Q \wedge R$ es más fuerte que Q y que R , y $P \vee R$ 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 precondition, 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 l entre k , ¿pero sabemos dividir?... Utilizaremos y ejercitaremos la técnica de diseño descendente en los ejemplos que iremos presentado en las notas.

Ejemplo de aplicación de diseño descendente:

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.

