# CALCULO DE PROGRAMAS

## Una metodología precisa para el diseño de programas de computador

*Jaime A. Bohórquez V.*
Universidad de Los Andes
Bogotá - *Colombia*

Resumen: Se presenta la metodología de diseño de programas de Gries y Dijkstra desde el punto de vista de estrategias de Solución de problemas. Se ilustra esta visión desarrollando completamente un ejemplo.

Palabras Claves: calculo de programas, corrección total, refinamiento a pasos, programación estructurada, aserciones.

## 0. Introducción

La programación de computadores debe considerarse como una forma de solución de problemas. Más precisamente, dado un problema bien especificado, se quiere construír un programa que lo solucione. Esta relación no se debe perder de vista al desarrollar (construír) cualquier programa, y debe siempre tenerse en cuenta que los métodos de programación están emparentados con los métodos (constructivos) de resolución de problemas.

Entre los pioneros en el desarrollo de una metodología de programación se encuentra Wirth [W71], quien es uno de los propulsores del método de refinamiento a pasos. Tales ideas son más ampliamente precisadas y fundamentadas por Hoare [H69], Dijkstra [Dj76] y Gries [G81] quienes proponen un cálculo de programas "guiado por objetivos" que consiste en una construcción simultánea del programa y la prueba de su corrección.

Desde otro punto de vista se encuentra a Jackson [J75], quien muestra las ventajas obtenidas al hacer corresponder la estructura de un programa con la de sus datos.

Aunque la presentación y estilo de estos autores es bien diferente, lo que en principio podría sugerir que se trata de enfoques independientes, en realidad existe una gran coherencia en sus trabajos. En la metodología de programación sugerida por Gries se reconoce una correspondencia entre las estructuras de control primitivas de un lenguaje algorítmico con las estrategias de solución de problemas utilizadas para diseñar programas; en el desarrollo de Jackson se encuentra una correspondencia análoga entre las mismas estructuras de control utilizadas por Gries y Dijkstra y las estructuras de datos que deben manejar los progra-

mas.

En este artículo se presenta el lenguaje algorítmico LCG (Lenguaje de Comandos Guardados) de Dijkstra como un instrumento para implementar una estrategia general de solución de problemas. La conexión del lenguaje LCG con estrategias de solución de problemas se entiende entre líneas en la literatura, pero el autor no conoce ninguna publicación en donde sea presentada explícitamente y con la importancia que se merece. (Algo en este sentido se encuentra en Hehner [Hh79], aunque allí la atención se centra en la proposición de un lenguaje algorítmico que excluye el comando iterativo, tal vez la pieza central del desarrollo de programas en LCG.)

Se hará énfasis en la aplicación práctica de una metodología de solución de problemas como herramienta de programación. La semántica de los comandos será introducida informalmente. Formulaciones precisas de los comandos LCG y su significado (semántica axiomática) se encuentran en [H69] y [Dj76].

## 1. El calculo de programas de Dijkstra y Gries.

La metodología de Dijkstra y Gries para el diseño de programas puede interpretarse en una estrategia de solución de problemas. Esta consiste inicialmente en la especificación del resultado deseado como una aserción formulada en principio en el cálculo de predicados (muchas veces es suficiente una enunciación precisa en lenguaje natural).

Obtenidas las pre- y post-condiciones que describen el problema, el lenguaje de programación se usa como un conjunto de comandos que efectuan transformaciones sobre predicados, que se denotan de la siguiente manera

$$\{Q\}\ S\ \{R\}$$

(léase el programa S es totalmente correcto con respecto a Q y R)

En esta expresión Q representa una precondición que es cierta antes de la ejecución del comando S, y R representa una post-condición que es verdad después de la ejecución del comando S. Es decir la formula significa que si la ejecución de S inicia en un estado que satisface Q, entonces terminará en un estado que satisface R.

La semántica de los comandos del lenguaje de programación concebido por Dijkstra está en correspondencia con tres métodos de solución de problemas:

a) Dividir y Conquistar (Comando de secuenciación)
   Dada la precondición Q y el objetivo R, establezca primero un subobjetivo $R_1$ mediante un comando (o programa) $S_1$ es decir

$$\{Q\}\ S_1\ \{R_1\}$$

luego, basandose en el logro de $R_1$, mediante un segundo comando $S_2$ establezca R, es decir $S_2$ es tal que

$$\{R_1\}\ S_2\ \{R\}$$

es cierto. Por tanto el programa que soluciona el problema original es la secuenciación de los comandos $S_1$ y $S_2$ o sea

$$S = S_1;S_2$$

b) Reducción a casos (comando de Selección)
Si se puede encontrar predicados $B_1$, $B_2$,..., $B_n$ tales que Q implique alguno de los $B_i$s, es decir $Q \Rightarrow B_1$ o $B_2$ o ... o $B_n$, entonces para establecer R dada la precondición Q basta usar el comando de selección

$$\begin{array}{lll} IF = \underline{if} & B_1 & ---> S_1 \\ \quad [] & B_2 & ---> S_2 \\ & \multicolumn{2}{c}{...} \\ \quad [] & B_n & ---> S_n \\ \underline{fi} & & \end{array}$$

válido cuando los $B_i$s son expresiones booleanas llamadas guardas, y los $S_i$s son comandos que cumplen

$$\{Q \text{ y } B_i\}\ S_i\ \{R\}$$

(Observe que las precondiciones de estos subobjetivos son más fuertes que la precondición original del programa).

El rectángulo "[]" sirve para separar alternativas no ordenadas.
Al entrar a ejecutar a IF, se escoge nodeterministicamente una guarda verdadera y se ejecuta el comando correspondiente.
La expresión $B_1$ ---> $S_1$ se puede leer como "en caso de que la guarda $B_1$ sea verdad se debe ejecutar el comando $S_1$".
Si ninguna guarda fuera verdadera para el estado inicial de ejecución de IF, el comando aborta.

c) Iteración (solución por transformación gradual)
El comando iteración tiene la forma

$$\begin{array}{lll} DO = \underline{do} & B_1 & ---> S_1 \\ \quad [] & B_2 & ---> S_2 \\ & \multicolumn{2}{c}{...} \\ \quad [] & B_n & ---> S_n \\ \underline{od} & & \end{array}$$

La iteración dura mientras por lo menos una de las guardas sea verdad. Del conjunto de comandos con guardas verdaderas se selecciona (no determinísticamente) una para ejecutarla.

Cuando se decide solucionar un problema con precondición Q y objetivo (o post-condición) R mediante una iteración, en general se tiene una idea o plan aunque sea vago para lograrlo. Estas ideas son plasmadas y concretadas en una aserción o predicado P que en general expresa un logro parcial del objetivo del problema.

El predicado P se llama un invariante del ciclo iterativo y los pasos para construir la iteración son los siguientes:

1) Partiendo de un estado cualquiera que cumpla Q (la precondición), obtener mediante un subprograma (en general trivial) un estado que satisfaga P es decir, establecer

{Q} inic {P}

donde inic es el subprograma mencionado.

2) Comparar el invariante y el objetivo para dilucidar una condición B que junto con el invariante implique la postcondición R es decir, encontrar B tal que

P y B => R

La negación de esta condición (¬B) constituirá en general, la única guarda del ciclo.

3) Escribir el cuerpo del ciclo de tal manera que su ejecución preserve la verdad del invariante y a la vez progrese hacia el logro de la guarda o condición B (y por consiguiente el logro de R).

El siguiente esquema resume los 3 pasos anteriores:

```
{Q}
inic
{P}
do ¬B ---> {¬B y P}
             S
           {P}
od
{R}
```

Aquí S es el cuerpo del ciclo.
Es posible que en la tarea de mantener a P invariante durante la ejecución de la iteración, sea necesario considerar varios casos; es decir, descomponer ¬B en subcondiciones $B_1$, $B_2$, ..., $B_n$ dando lugar a la forma general del comando de iteración DO.

Para garantizar la terminación de la ejecución del ciclo iterativo, se usa una función cota t (entera) que depende del estado de ejecución, y estima el número de iteraciones que aún deben ejecutarse para terminar el comando.

La función cota t debe cumplir las siguientes propiedades:

(i) P y B => t>0
es decir mientras dure el ciclo t debe ser positivo.

(ii) {P y B y t≤ß} S {t<ß}
es decir t debe decrecer con cada iteración.

Por lo anterior, t decrece y se mantiene positiva mientras dura el ciclo iterativo, debido a ésto el ciclo no puede prolongarse indefinidamente (el rango de t es bien fundado); ß es un valor constante y entero que no aparece en ninguna otra parte del programa.

Esta última condición (ii) constituye una guía para el diseño del cuerpo del ciclo en el paso (3): Mantener el invariante progresando hacia terminación.

Se tiene entonces, un lenguaje de programación que responde a una metodología de solución de problemas con tres estrategias básicas Dividir y Conquistar, Reducción a casos y Transfomación gradual apoyada en una propiedad invariante; las cuales corresponden a las operaciones básicas de secuenciación, selección e iteración que son aplicadas recurrentemente a los subproblemas que van surgiendo con su aplicación, hasta llegar a subproblemas de solución sencilla (†).

Se puede también añadir a las tres estrategias mencionadas arriba, la que corresponde a la acción de abstraer la solución de un problema como un subprograma o procedimiento, posponiendo su solución a una etapa posterior del diseño. De hecho, Hehner [Hh79] propone la abstracción de subproblemas como herramienta principal del desarollo de programas, como al alternativa a la estrategia de transformación gradual mencionada en el párrafo anterior.

Es interesante el hecho de que Jackson [J75] proponga esta misma trilogía para el tratamiento de estructuras de datos secuenciales logrando de esta manera programas que corresponden estructuralmente a la jerarquía semántica de sus datos. Mediante este mecanismo se obtienen programas muy facilmente modificables para mantenerse al día con la evolución y cambios requeridos por el usuario del programa.

Queda ilustrado de esta manera el "isomorfismo" que existe entre el lenguaje de comandos guardados y la estrategia de solución de problemas descrita anteriormente. Cabe agregar que los comandos de dicho lenguaje son versiones generalizadas de los usados por casi todos los lenguajes imperativos de programación y coinciden con los recomendados por las disciplina informal comunmente denominada programación estructurada.

---

(†) Como comando primitivo de transformación de estados se tiene por supuesto a la asignación; cuyos efectos se pueden describir en el calculo que se está presentando como {Q} x:=e {R} si y solamente si $Q \Rightarrow R^x_e$. Donde $R^x_e$ denota la sustitución textual de cada instancia libre de x por la expresión e.

Estos hechos contrastan con los métodos tradicionales de la enseñanza de la programación que reducen la labor de programar al diestro manejo y comprensión de los diferentes comandos de un lenguaje de programación, en total desconección con una disciplina de solución de problemas; juzgando además, la bondad de un lenguaje de programación por la variedad y sofisticación de sus comandos básicos como factor preponderante.

## 2. El cálculo en la práctica.

A continuación se mostrará el uso de la metodología previamente descrita construyendo un programa para solucionar un problema de dificultad moderada que permitirá sin embargo, ilustrar cada uno de los puntos discutidos anteriormente.

El siguiente problema fué tomado de [W182].

Problema: Dados n enteros mayores que 1, (n entero positivo) calcular para cada uno de los datos sus primos más cercanos (Si dos primos son equidistantes a alguno de los datos ambos deben ser calculados.)

Especificación mediante aserciones

Precondición: Q: d(1),...,d(n) son enteros mayores que 1.
Postcondición: R: C(1),...,C(n) son conjuntos de números tales que C(i)={p: p es primo más cercano a d(i)} para i=1,...,n.

Construcción del programa

La solución (global) evidente para este problema es un ciclo iterativo cuyo invariante en la etapa i reza

P: $1 \leq i \leq n$ y C(j)={p: p es primo más cercano a d(j)} para j=1,..,i-1

Para no aburrir al lector con detalles de la aplicación de la metodología en esta primera fase, que es bastante evidente, se presenta sin más explicaciones la primera aproximación a la solución del problema.

```
i:=1;
do i≠n+1 ---> CalcPrimosCercanos(d(i),C(i));
              i:=i+1
od
```

CalcPrimosCercanos(d,C) es un subprograma cuyo texto está por escribirse (esta acción corresponde a la aplicación de una estrategia de abstracción); que recibe un entero d mayor que 1 como argumento, y deja en C el conjunto de primos más cercanos a éste.

Una vez calculado el texto de este subprograma el problema estará resuelto.

Antes de continuar aplicando la metodología a dicho subproblema conviene notar que la corrección del desarrollo que se tiene hasta ahora, es trivialmente verificable:

-P vale trivialmente después de la asignación i:=1
-P y no-(i≠n) (i.e. P y i=n) implican la postcondición R
-{P y i≠n} CalcPrimosCercanos(d(i),C(i)); i:=i+1 {P}

(Si CalcPrimosCercanos se comporta en la forma descrita arriba la terminación del problema es evidente.

Desarrollo del subprograma CalcPrimosCercanos

Este subproblema puede especificarse así:

Precondición $Q_1$: d es un entero mayor que 1
Postcondición $R_1$: C = {p: p es primo más cercano a d}

Se puede reducir la solución a dos casos:
d es primo y d no es primo (donde en el primer caso la solución es inmediata). Usando el comando de selección se obtiene

{$Q_1$}
if d es primo ---> C:={d}
[] d no es primo ---> $S_1$
fi
{$R_1$}

Aquí {d} es el conjunto cuyo único elemento es d y $S_1$ es un subprograma que debe cumplir la especificacion

$$\{Q_1 \text{ y d no es primo}\}\ S_1\ \{R_1\}$$

Para obtener $S_1$ se observa que la postcondicion $R_1$ se puede escribir así:

$R_1$: $R_1'$ y C = {p: p es primo, y $p=p_1$ ó $p=p_2$} donde

$R_1'$: -|---------|----------|----------|---> , $p_1$ o $p_2$ es primo y
0 $p_1$ d $p_2$
no existe ningún primo estrictamente entre $p_1$ y $p_2$.
La gráfica de arriba ilustra la situación relativa de $p_1$ y $p_2$ abreviando la formula "$p_1$ y $p_2$ son equidistantes a d y $p_1<p_2$".

Si se aplica ahora, la técnica de "dividir y conquistar" obteniendo primero a $R_1'$ y luego a $R_1$ mediante la secuenciación $S_{11};S_{12}$ de tal forma qué $S_{11}$ y $S_{12}$ cumplan respectivamente

$$\{Q_1 \text{ y d no es primo}\}\ S_{11}\ \{R_1'\} \quad \text{y} \quad \{R_1'\}\ S_{12}\ \{R_1\}$$

como $R_1$ => R, esto es suficiente para lograr el objetivo propuesto.

$S_{12}$ es bastante sencillo y no amerita reflexiones adicionales:

```
if p1 es primo      ---> C:={p1}
[] p1 no es primo   ---> C:={}
fi;
if p2 es primo      ---> C:=C U {p2}
[] p2 no es primo   ---> skip
fi
```

{} denota el conjunto vacío, skip es un comando que no afecta ninguna variable (i.e. deja las cosas como estan).
Mucho más interesante es la construcción de $S_{11}$. Esta se logra mediante un ciclo iterativo cuyo predicado invariante consiste en un "logro parcial" de $R_1$:

$P_1$: -|----|----|-- > y no existe ningún primo estrictamente
$p_1$ d $p_2$
entre $p_1$ y $p_2$.

Observe que la inicialización $p_1:=d$; $p_2:=d$ cumple el invariante $P_1$, aunque se puede ser más sutil:

```
if d es par   ---> p1:=d-1; p2:=d+1
[] d es impar ---> p1:=d-2; p2:=d+2
fi
```

La precondición de $S_{11}$ garantiza que d no es primo. Además como $P_1$ y ($p_1$ es primo o $p_2$ es primo) implican $R_1$, la guarda para el ciclo en construcción es: $p_1$ no es primo y $p_2$ no es primo; obteniendose

```
{Q1 y d no es primo}
if d es par   ---> p1:=d-1;p2:=d+1
[] d es impar ---> p1:=d-2;p2:=d+2
fi;
{P1}
do ¬primo(p1) y ¬primo(p2) ---> "Preservar el invariante
                                 P1 incrementando la
                                 longitud del intervalo
                                 [p1,p2]"
od
```

Se puede pensar en primo(p) como una función lógica que habla sobre la verdad de la afirmación "p es primo".
La preservación del invariante para el cuerpo del ciclo se logra mediante el comando $p_1:=p_1-2$; $p_2:=p_2+2$. La corrección de este paso así como la terminación del ciclo estan garantizadas por propiedades elementales de los números primos.

En resumen se tiene

```
i:=1;
do i≠n+1 --->
    if primo(d(i))   ---> C:={d(i)}
    ▯ ¬primo(d(i)) ---> if d(i) es par   ---> p1:=d(i)-1;p2:=d(i)+1
                        ▯ d(i) es impar ---> p1:=d(i)-2;p2:=d(i)+2
                        fi;
                        do ¬primo(p1) y ¬primo --->
                                        p1:=p1-2;p2:=p2+2
                        od;
                        if primo(p1)  ---> C(i):={p1}
                        ▯ ¬primo(p1)  ---> C(i):={}
                        fi;
                        if primo(p2)  ---> C(i):=C(i) U {p2}
                        ▯ ¬primo(p2)  ---> skip
                        fi
    fi;
    i:=i+1
od
```

El desarrollo mostrado hasta este punto del programa es suficiente para dar una idea más o menos precisa de la propuesta matodologica que se pretendía exponer. Sin embargo el lector interesado podrá estudiar el diseño ulterior de la función decisora primo(p) en el apéndice al final de este artículo.

## 3. Conclusiones

El ejemplo anterior ha permitido ilustrar la correspondencia anunciada en la introducción, entre el lenguaje algorítmico usado, y las tres estrategias de solución de problemas ya mencionadas. Se ha visto también como la construcción del programa y la prueba de su corrección pueden "ir de la mano"; con la prueba de la corrección guiando la construcción.

Como subproducto de hacer de la programación una actividad "guiada por objetivos" mediante la especificación de cada etapa de la construcción del programa por medio de aserciones, se obtienen programas a la vez, claros, sucintos y elegantes.

Los métodos presentados en este artículo son de naturaleza deductiva, es decir van de lo global a lo particular. Para una propuesta metodológica basada en métodos inductivos ver Dromey [Dr85].

## APENDICE

Se continuará aquí, la aplicación de la metodología de Dijkstra y Gries al problema en cuestión, diseñando una función que decida si un entero positivo es primo o no.
Tomando prestado un poco de la notación del lenguaje Pascal tendriamos:

```
function primo(d: nat): boolean;
{Q2: d>0}
{R2:   (primo(d)=true  y d es primo)
     o (primo(d)=false y d no es primo)}
```

Para la elaboración de un plan de ataque al problema conviene estudiar la definición de número primo.

Un entero positivo d es primo si y solo si

(1) d≠1 y los únicos divisores positivos de d son 1 y d mismo.

Esta definición puede establecerse en forma positiva y acotada como

d≠1 y todo entero k que cumpla $2 \leq k < d$ no es un divisor de d.

Esta definición sugiere un estudio de casos y un ciclo iterativo para decidir si d es primo. Sin embargo se puede hilar más fino:

Un entero positivo d es primo si y solo si

(2) d=2 o (d es un impar mayor que 1 tal que todo número impar que cumpla $3 \leq k \leq [d]$ no es divisor de d)

( [x] representa la parte entera (mayor entero menor que) del número x )

Se interpretará entonces la noción de primo de acuerdo a la definición (2)

Una reducción de casos obtendría el siguiente comando

```
{d>0}
if d=2                    ---> primo(d):= true
[] d>1 y d impar          ---> S3
[] d=1 o (d par y d>2)    ---> primo(d):= false
fi
{R2}
```

Donde $S_3$ es un subcomando que debe cumplir:

$\{d>1 \text{ y } d \text{ impar}\}\ S_3\ \{R_2\}$

$S_3$ se diseña mediante un ciclo iterativo cuyo predicado invariante se obtiene debilitando la aserción $R_2$ a $P_2$:

$P_2$: d>1, d impar y además i es un impar con $3 \leq i \leq [\,d\,]+2$ que cumple

(primo(d)=true y $\forall$k(k impar y $3 \leq k < i$ => k$\not|$d))

o (primo(d)=false, $i \leq [\,d\,]$ y i|d).

Es decir, i es una variable de trabajo que variará sobre los números impares desde 3 hasta máximo [ d]+2 para la cual todos los impares estrictamente menores que ella no dividen a d y el valor de primo(d) es true; o el valor de p(d) es false, e i es un "testigo" de la no primalidad de d, por el hecho de dividirlo.

Por tanto $P_2$ y (primo(d)=false o i>[ d]) implican $R_2$, es decir, (primo(d)=true y d es primo) o (primo(d)=false y d no es primo).

Además t=[ d]-i es una cota superior para el número de iteraciones que aun queden por ejecutar en el siguiente ciclo:

```
{d>1 y d impar}
i:=3;
primo(d):=true;
{P2}
do i≤[ d] y primo(d)=true --->
            if i|d ---> primo(d):=false
            [] i/|d ---> i:=i+2
            fi
od
{R2}
```

## BIBLIOGRAFIA

[Dj76] Dijkstra, E.W.: A Discipline of Programming. New Jersey: Prentice-Hall 1976

[Dr85] Dromey, R.G.: Program Development by Inductive Stepwise Refinement. Software Practice and Experience, Vol. 15(1), pp 1-28, Enero 1985

[G81] Gries, D.: The Science of Programming. Springer Verlag, N.Y., 1981

[H69] Hoare, C.A.R.: An Axiomatic basis for Computer Programming. CACM, 12, pp 576-80, 1969

[Hh79] Hehner, E.C.R.: **do** Considered **od**: A Contribution to the Programming Calculus. Acta Informatica 11, pp 287-304, 1979

[J75] Jackson, M.A.: Principles of Program Design. Academic Press, London, 1975

[W71] Wirth, N.: Program Development by Stepwise Refinement, CACM 14(4) pp. 221-227, Abril 1971

[W182] Welsh, J., Elder, J.: Introduction to Pascal. Prentice Hall, Inc. Englewood Cliffs, N.J. 1979