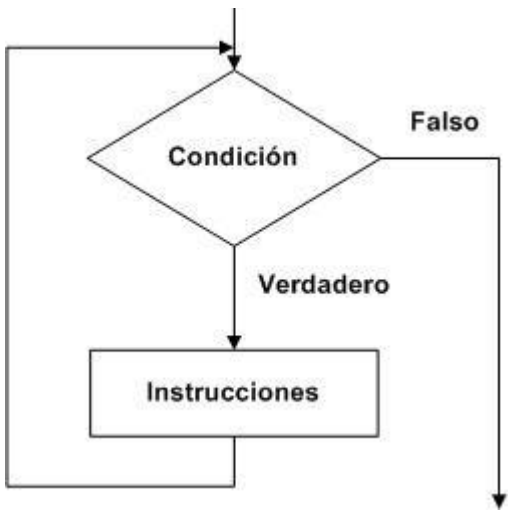
Capítulo 11
El comando iterativo

El ciclo while convencional y el comando iterativo

El bucle while en Pascal tiene la forma "while B do S" y en PL/I la forma bastante barroca "DO WHILE (B); S END;" para un booleano B y comando S. S a veces se llama el cuerpo del loop. La ejecución del while-loop se puede expresar usando un goto estado como:

loop: **`if B then begin S; goto loop end`**

pero a menudo se describe mediante un diagrama:



En nuestra notación de programación, el ciclo while tiene la forma

do B -> S od

donde B -> S es un comando guardado. Llamamos el comando iterativo y al que nos referimos por el nombre DO.

```
do B1 -> S1
 | B2 -> S2
 .
 .
 .
 | Bn -> Sn
od
```
(11.1)

donde n > 0 y cada B_i -> S_i es un comando guardado. Tenga en cuenta la similitud de sintaxis entre DO y IF; uno es un conjunto de comandos guardados encerrados en do y od, el otro un conjunto encerrado en if y fi.

En una oración, así es como se puede ejecutar (11.1).

Repetir (o iterar) lo siguiente hasta que ya no sea posible: elegir un protector B_i que sea true y ejecute el comando correspondiente S_i.

Al terminar todas las guardias son falsas.

Elegir una guarda verdadera y ejecutar su comando se llama realizar una iteración del ciclo.

Tenga en cuenta que se permite el no determinismo: si dos o más guardias son verdaderas, se elige cualquiera (pero sólo uno) y se ejecuta el comando correspondiente en cada iteración. Usando IF para denotar el comando alternativo con los mismos comandos guardados y BB para denotar la disyunción de las guardas (ver capítulo 10), vemos que (1 1.1) es equivalente a

```
do BB -> if B1 -> S2
           | B2 -> S2
           .
           .
           .
           | Bn -> Sn
         fi
od
```

o

```
do BB -> IF od
```

Es decir, si todas las guardias son falsas, lo que significa que BB es falsa, la ejecución termina; de lo contrario, se ejecuta el correspondiente comando alternativo IF y se repite el proceso. Una iteración de un bucle, por lo tanto, es equivalente a encontrar BB verdadero y ejecutar IF.

Por lo tanto, podemos arreglárnoslas solo con el simple while-loop. Sin embargo, nosotros seguiremos usando la forma más general porque es

extremadamente útil en el desarrollo de programas, como veremos en la Parte III.

La definición formal de DO

El siguiente predicado $H_0(Q)$ representa el conjunto de estados en los que la ejecución de DO termina en 0 iteraciones con Q verdadero, porque las guardas son inicialmente falsos:

$$H_0(Q) = \text{NOT BB AND } Q$$

Escribamos también un predicado $H_k(Q)$, para $k > 0$, para representar el conjunto de todos los estados en los que la ejecución de DO termina en k iteraciones o menos, con Q cierto.

La definición será recursiva, es decir, en términos de $H_{k-1}(Q)$.

Un caso es que DO termina en 0 iteraciones, en cuyo caso $H_0(Q)$ es verdadera.

El otro caso es que se realiza al menos una iteración. Por lo tanto, BB debe inicialmente ser verdadero y la iteración consiste en ejecutar el IF correspondiente.

Esta ejecución de IF debe terminar en un estado en el cual el ciclo va a iterar k - 1 o menos veces. Esto lleva a

$$H_k(Q) = H_0(Q) \text{ OR } WP(IF, H_{k-1}(Q)), \text{ para } k > 0.$$

Ahora, WP (DO, R) representa el conjunto de estados en los que la ejecución de DO termina en un número limitado de iteraciones con Q verdadero. Eso es, inicialmente debe haber algún k tal que como máximo se realicen k iteraciones. Por lo tanto, definimos:

(11.2) Definición. $WP(DO, Q) = (\text{Existe k: Z})(0 <= k \text{ AND } H_k(Q))$ ❑

Dos ejemplos de razonamiento sobre bucles

La definición formal de DO no es fácil de usar y no da una idea en el desarrollo de programas. Por lo tanto, queremos desarrollar un teorema que nos permita trabajar con una precondición útil de un bucle (con respecto a un post condición) que no es la precondición más débil.

Ilustramos primero la idea con dos ejemplos:

Se supone que la ejecución del siguiente algoritmo es almacenar en la variable s la suma de los elementos del arreglo b [0:10].

```
i , s : = 1 , b [ 0 ]
d o  i < 1 1  - >  i , s : = i + 1 , s + b [ i ] o d
{ Q :  s  = ∑_0^10 b[k] }
```

¿Cómo podemos argumentar que funciona? Comencemos dando un predicado I que muestra la relación lógica entre las variables i , s y b. En efecto, esto sirve como una definición de i y s:

$$\text{I: } 1 \le i \le 11 \ \wedge \text{s} = \ \sum_0^{i-1} b[k]$$

Mostraremos que I es cierto justo antes y después de cada iteración del bucle, por lo que también es cierto al terminar.
Si I es cierto en todos estos lugares, entonces, con la ayuda adicional de la falsedad de los guardias, podemos ver que Q también es cierto en la terminación (ya que $I \wedge i \geq 11 \rightarrow Q$). Resumimos lo que necesitamos mostrar anotando el algoritmo:

```
{ T }
i , s : = 1 , b [ 0 ]
{ I }
d o  i < 1 1  - > { i < 11 ∧ I } i , s : = i + 1 , s + b [ i ] { I } o d
{ i ≥ 11 ∧ I }
{ Q }
```

Repetimos, porque es ¡muy importante!:

Si podemos demostrar que (1) I es verdadero antes de la ejecución del bucle y que (2) cada iteración del bucle sale I verdadero, entonces I es verdadero antes y después de cada iteración y al terminar. Entonces, la verdad de I y la falsedad de la guarda nos permiten deducir que se ha establecido el resultado deseado Q
Ahora vamos a verificar que I es verdadera después de la inicialización `i , s : = 1 , b [ 0 ]`, no importa cuál sea el estado inicial.
Podemos ver esto informalmente, o podemos probarlo de la siguiente manera:

$$\text{WP (“i, s:=1, b[0]”, I)} = 1 \le 1 \le 11 \ \wedge b[0] = \sum_0^0 b[k]$$
$$= T.$$

Ahora vamos a mostrar que una iteración del ciclo termina con I verdadero, es decir una ejecución del comando `i , s : = i + 1 , s + b [ i ]` comenzando con I e $i < 11$ verdadero termina con I todavía verdadero. Nuevamente, podemos ver esto de manera informal, o podemos demostrarlo formalmente:

$$\text{WP (“i, s:= i+1”, s+ b[i], I)} = 1 \leq i + 1 \leq 11 \ \wedge s + b[i] = \sum_0^i b[k]$$
$$= 0 \leq i < 11 \ \wedge s = \sum_0^{i-1} b[k]$$

y $(I \wedge i < 11)$ implica la última línea.

Por lo tanto, sabemos que si la ejecución del ciclo termina, al terminar $(I \wedge i \geq 11)$, y por lo tanto Q, son verdaderas.

Un predicado I que es verdadero antes y después de cada iteración de un bucle es llamada relación invariante, o simplemente invariante, del bucle. (El adjetivo invariante significa constante o inmutable en matemáticas el término significa no afectado por el grupo de operaciones matemáticas bajo consideración, la única operación aquí es una iteración del ciclo bajo la verdad inicial de P).

Para mostrar que el ciclo termina, introducimos una función entera, t, de las variables del programa que es un límite superior en el número de iteraciones aún por realizar. Cada iteración del bucle disminuye t en menos 1 y, siempre que la ejecución del ciclo no haya terminado, t es acotado abajo por 0. Por lo tanto, el ciclo debe terminar. Sea t:

```
t: 11 - i
```

Dado que cada iteración aumenta i en 1, obviamente disminuye t en 1. Además, siempre que haya una iteración que realizar, es decir, siempre que i < 11, sabemos que t es mayor que 0.

En este caso, t indica exactamente cuántas iteraciones quedan por realizar, pero en general, sólo puede proporcionar un límite superior en el número de iteraciones aún por realizar. La función t ha sido llamada variante, a diferencia de la relación invariante I la función cambia en cada iteración; la relación permanece invariablemente cierta. Sin embargo, para enfatizar su propósito, llamaremos t como la función ligada.

El ejemplo anterior puede parecer que requiere demasiada explicación. para un algoritmo tan simple. Consideremos ahora un segundo ejemplo, cuya corrección no es tan obvia. De hecho, es sólo con la ayuda de la invariante que seremos capaces de entenderlo.

El algoritmo (11.4) planteado consiste en almacenar en la variable z el valor a*b para b $\geq$ 0, pero sin el uso de multiplicación.

(11.4)

```
{P: b ≥ 0}
x, y, z := a, b, 0
do  y > 0 ∧ par(y)  -> y, x := y / 2, x + x
  | impar(y)        -> y, z := y - 1, z + x
od
{Q: z = a * b}
```

Una vista del ciclo es que procesa la representación binaria de b, que se ha almacenado en y. La prueba de imparidad y paridad se realiza interrogando el bit más a la derecha, restando 1 cuando el bit más a la derecha es 1, significa cambiarlo a cero, y dividir por 2 se hace desplazando la representación binaria 1 bit a la derecha, eliminando así el bit más a la derecha.

Pero, ¿cómo sabemos que el algoritmo funciona? Nosotros presentamos - sacando del sombrero mágico, por así decirlo -, el invariante I (cómo encontrar invariantes es un tema de Parte III):

$$\text{I: } y \geq 0 \ \wedge z + x * y = a * b.$$

Determinamos que I es verdadera justo después de la inicialización:

$$\text{WP ("x, y ,z := a, b, 0", I)} = b \geq 0 \ \wedge 0 + a * b = a * b,$$

lo cual obviamente está implícito en la condición previa del algoritmo (11.4). Próximo, mostramos que cualquier iteración del ciclo que comienza con I verdadero termina con I verdadero, por lo que I es un invariante del bucle. Para el segundo comando guardado, esto se puede observar que el valor de z + x*y permanece igual si y se reduce en 1 y x se suma a z:

$$z + x*y = z + x + x*(y - 1)$$

Para el primer comando guardado, tenga en cuenta que la ejecución de y, x := y/2, x + x con y par deja el valor de z + x*y sin cambios, porque x*y = (x + x)*(y/2) cuando y es par. Dejamos la verificación más formal para el lector (ejercicio 7).

Dado que cada iteración del ciclo deja I verdadero, I debe ser verdadero cuando termina. Mostramos que I junto con la falsedad de los guardias implica el resultado Q como sigue:

$I \wedge \sim (y > 0 \wedge par(y)) \wedge \sim impar(y)$
$= y \geq 0 \wedge z + x * y = a * b \wedge (y \leq 0 \wedge par(y))$
$= y = 0 \wedge z + x * y = a * b$
⇨ z = a*b

El trabajo realizado hasta ahora se transmite mediante el siguiente programa anotado.

(11.5)
```
{P : b ≥ 0}
x , y , z := a , b , 0
{I}
do  y > 0 ∧ par(y)  -> {I ∧ y > 0 ∧ par(y)} y , x := y / 2 , x + x {I}
 | impar(y)         -> {I ∧ impar(y)}       y , z := y - 1 , z + x {I}
od
{I ∧ y ≤ 0 ∧~ impar(y)}
{I ∧ y = 0}
{Q : z = a * b}
```

Para mostrar que el ciclo termina, use la función ligada t = y: es mayor que 0 si hay otra iteración para ejecutar y se reduce en al menos 1 en cada iteración.

Un teorema sobre un bucle, un invariante y una función ligada

En los dos ejemplos que acabamos de dar, se usó el mismo tipo de razonamiento para argumentar que los bucles funcionaron como se deseaba. Esta forma de razonamiento es incorporado en el teorema (11.6). A estas alturas, el teorema debería ser bastante claro.

La suposición 1 implica que I será verdadera al terminar DO.

El supuesto 2 indica que la función t está acotada por debajo de 0 siempre que la ejecución de DO no ha terminado.

El supuesto 3 indica que cada iteración disminuye t en al menos uno, por lo que se garantiza la terminación para ocurrir. Un número ilimitado de iteraciones disminuiría t por debajo de cualquier límite, lo que llevaría a una contradicción.

Finalmente, al terminar todas las guardas son falsas, por lo que BB es verdadera.

(11.6) Teorema. Considere el bucle DO. Supongamos que un predicado I satisface:

1. I $\wedge B_i \rightarrow WP\,(S_i, I)$, para todo i, $1 \leq i \leq n$.

Suponemos fuertemente, que la función entera t satisface lo siguiente, donde t1 es un identificador libre:

2. $I \wedge BB \rightarrow (t > 0)$
3. $I \wedge B_i \rightarrow WP\,(\text{t1} := \text{t}, \text{S}_\text{i}, \ t < t1)$, para $1 \leq i \leq n$.

Entonces $I \rightarrow WP\,(DO, \ I \wedge \sim BB)$

Prueba. Realizada en la teórica de ciclos e invariantes uba-aed1.

Discusión

Un bucle tiene muchos invariantes. Por ejemplo, el predicado x*0 = 0 es un invariante de cada bucle ya que siempre es verdadero. Pero un invariante que satisface los supuestos del teorema (11.6) es importante porque proporciona comprensión del bucle. De hecho, cada **ciclo, excepto el más trivial, debe anotarse con un invariante que satisfaga el teorema.**

Como veremos en la Parte III, el invariante no sólo es útil para el lector, es casi necesario para el programador. Daremos heuristicas para desarrollar la función invariante y ligada antes de desarrollar el loop y argumentan que esta es la forma más efectiva de programar. Este tiene sentido si vemos el invariante como simplemente la definición de las variables y recuerda el adagio acerca de definir con precisión las variables antes de ser utilizarlos.

En este punto, por supuesto, el desarrollo de un invariante puede parecer casi imposible, ya que incluso la idea de un invariante es nueva.

Deja el proceso de desarrollo a la Parte III, y por ahora concentrarse en entender los bucles para los que ya se proporcionan invariantes.

Anotar un bucle y comprender la anotación

Los algoritmos (11.3) y (1.5) están anotados para mostrar cuándo y dónde los invariantes son verdaderos. En lugar de escribir el invariante en tantos lugares, a menudo es más fácil dar la función invariante y ligada en el texto acompañando a un algoritmo.

Cuando sea necesario incluirlos en el algoritmo en sí, es ventajoso usar una abreviatura, como se muestra en (11.8).

```
{P}
{inv I: el invariante}
{bound t: la función variante}
do B1 -> S1
  | B2 -> S2
  .
  .
  .
  | Bn -> Sn
od
{Q}
```

(11.8)

Ante un lazo con forma (11.8), según el teorema (11.6) el lector solo necesita verificar los puntos dados en (11.9) para comprender que el bucle es correcto.

La existencia de una lista de verificación de este tipo es, de hecho, es una ventaja, ya que permite estar seguro de que no se ha olvidado nada. En realidad, la lista de verificación es útil para el propio programador, aunque después de un tiempo (juego de palabras) su uso se convierte en una segunda naturaleza.

(11.9) Lista de verificación para comprender un bucle:

1. Demuestre que I es verdadera antes de que comience la ejecución del ciclo.
2. Demostrar que $I \wedge B_i \rightarrow WP(S_{i,}\ I)$, para $1 \leq i \leq n$. Es decir, la ejecución de cada comando protegido termina con I verdadero, por lo que I es en efecto un invariante del loop.
3. Demuestre que $I \wedge \sim BB \rightarrow Q$, es decir, al terminar el deseado resultado es verdadero.
4. Demostrar que $I \wedge BB \rightarrow (t > 0)$, de modo que t está delimitado por abajo siempre y cuando el bucle no haya terminado.
5. Demuestra que $\{I \wedge B_i\}\, t1 \coloneqq t, S_i\, \{t < t1\}$, para $1 \leq i \leq n$, de modo que se garantiza que cada iteración del bucle disminuirá la función vinculada.

A menudo, solo es necesario proporcionar la función invariante y ligada como documentación para un bucle, porque el algoritmo es casi trivial para verificar.

Esta es la mejor documentación: sólo lo suficiente para proporcionar la comprensión necesaria y no tanto que el lector se pierda en superfluitos y obviedades.

En la misma línea, las partes de un invariante que se refieren sólo a invariable.
Las variables de un ciclo a menudo se omiten, y se espera que el lector observe este.

Por ejemplo, en el algoritmo (11.3) no indicamos explícitamente que b se mantuvo sin cambios (al incluir como un conjunto de la pre condición, el invariante y la pos condición el predicado b = B donde B representa el valor inicial de b. De manera similar, en el algoritmo (5.4) no hicimos indicar explícitamente que a y b permanecieron sin cambios.

Es importante realizar varios de los ejercicios 7-13. En la Parte III nos discutiremos el desarrollo de bucles, pero la Parte III tendrá sentido y parece fácil solo si está completamente familiarizado con el teorema 11.6 y el uso de la lista de verificación 11.9 y si ha ganado alguna facilidad de esta manera de pensar.

Ejercicios para el Capítulo 11

1. Ver el enunciado en la página 146 del libro the science of programming y las soluciones más abajo.