Departamento de Informática | Universidad de Valladolid

Programación II (I.T.I. de Gestión)

Verificación de algoritmos

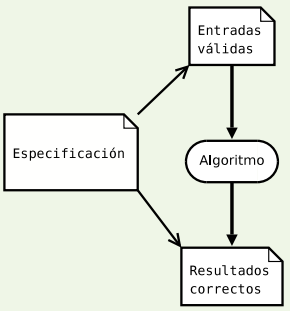
Félix Prieto
Esperanza Manso

Curso 2009/10

Introducción

- Las técnicas de verificación de programas no persiguen aumentar la fiabilidad del código, sino *demostrar* que no contiene errores
- La verificación...
  - Mejora nuestra comprensión del código
  - Facilita la comunicación del código
  - Puede aumentar la fiabilidad, aunque es difícil hacer demostraciones completas
  - Con las herramientas adecuadas puede integrarse en el código

El papel de la especificación



- Necesitamos describir entradas válidas
- Necesitamos describir resultados correctos para cada entrada
- Las descripciones deben ser formales si queremos demostrar

Elementos usados en verificación

- *Variables ligadas*: Son las variables asociadas a un cuantificador. En expresiones como $\forall \epsilon > 0$, $\exists \delta > 0$ o $\sum_{i=1}^{10} a_i$, ϵ, δ e i son variables ligadas
- *Variables libres iniciales*: Representan el valor original de los datos de entrada del programa o valores constantes. Las variables libres iniciales tienen un valor fijo, no necesariamente conocido durante la demostración
- *Variables libres del programa*: Son las variables utilizadas en el código de nuestro algoritmo o programa. El valor de las variables libres del programa suele venir dado por una expresión en función de las variables libres iniciales, utilizando en su caso variables ligadas, y suele variar durante la ejecución del código

Elementos usados en verificación (II)

- *Aserto*: Expresión lógica escrita mediante variables libres iniciales, variables libres del programa y variables ligadas a un cuantificador
- Los asertos permiten expresar información sobre el estado del entorno de un programa o algoritmo y con ello *demostrar* sobre su funcionamiento
- Llamamos *precondición* para un algoritmo a un aserto con información previa a la ejecución del código
- Llamamos *postcondición* para un algoritmo a un aserto con información sobre el resultado de la ejecución del código
- El par dado por una precondición y una postcondición forman una *especificación pre-post* para un algoritmo

Elementos usados en verificación (III)

algoritmo

$$a, b, c \in \mathbb{R}, a > 0 \quad (1)$$

$$(2)$$

$$\forall i \in \{1, 2\} \;\; a x_i^2 + b x_i + c = 0 \quad (3)$$

- a, b, c son variables libres iniciales
- x_1, x_2 son variables libres del programa
- i es una variable ligada
- (1) actúa como precondición para el algoritmo
- (3) actúa como postcondición para el algoritmo

Elementos usados en verificación (IV)

- Dados dos asertos que verifican $A_1 \implies A_2$, de modo informal diremos que:
  - A_1 es *más fuerte* que A_2, o alternativamente que
  - A_2 es *más débil* que A_1
- El aserto más fuerte de todos es «Falso»
- El aserto más débil de todos es «Cierto»

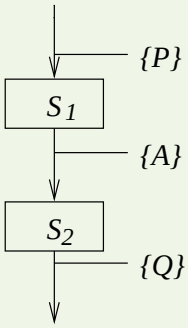
Definiciones fundamentales

- Definición 1 Un algoritmo es *parcialmente correcto* respecto de una especificación pre-post si siempre que empieza en condiciones que satisfacen la precondición *y termina*, lo hace en condiciones que satisfacen la postcondición
- Definición 2 Un algoritmo es *totalmente correcto* respecto de una especificación pre-post si siempre que empieza en condiciones que satisfacen la precondición, *termina* y lo hace en condiciones que satisfacen la postcondición
- Un algoritmo parcialmente correcto es totalmente correcto si es finito cuando comienza en condiciones que satisfacen la precondición

Comentarios sobre las definiciones

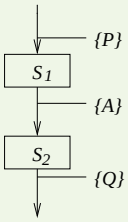
- Todo programa infinito es parcialmente correcto por definición
- Todo programa es totalmente correcto cuando utilizamos como precondición «Falso»
- Todo programa es parcialmente correcto cuando utilizamos como postcondición «Cierto»
- Todo programa finito es totalmente correcto cuando utilizamos como postcondición «Cierto»
- «Cierto» y «Falso» pueden ser escritos de formas muy complicadas
- Demostrar que un algoritmo es parcial o totalmente correcto es fácil si nos dejan fijar la especificación
- La corrección parcial sólo es útil como paso intermedio en la demostración de la corrección total

Método básico de demostración



Para demostrar la corrección (parcial o total) de $\{P\}S_1; S_2\{Q\}$ buscamos un aserto intermedio para el que sea fácil demostrar la corrección (parcial o total) de $\{P\}S_1\{A\}$ y $\{A\}S_2\{Q\}$

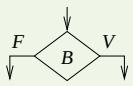
Método básico de demostración (II)



- Hacemos *derivación directa* cuando conocemos S_1, S_2 y P y debemos obtener Q tan fuerte como sea posible para hacer el conjunto totalmente correcto
- Hacemos *derivación inversa* cuando conocemos S_1, S_2 y Q y debemos obtener P tan débil como sea posible para hacer el conjunto totalmente correcto
- Hacemos *programación sistemática* cuando conocemos P y Q y debemos obtener S_1 y S_2
- Sin embargo es habitual plantear ejercicios en que se conocen P, Q, S_1 y S_2

Tipos de nodos posibles en un algoritmo

Unión



Decisión

Asignación

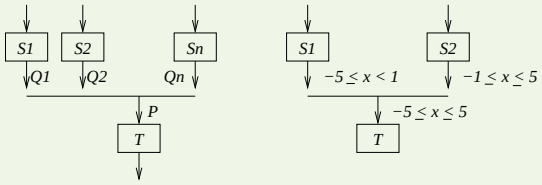
- Los algoritmos se pueden describir mediante diagramas de flujo utilizando sólo tres tipos de nodos:
  - Nodos de unión
  - Nodos de decisión
  - Nodos de asignación
- En los nodos de asignación v es una variable libre del programa, mientras que w representa una expresión
- Deberemos establecer reglas de demostración sobre cada uno de estos tipos de nodo para demostrar sobre cualquier algoritmo

Verificación de los nodos de unión

Regla 1 Un nodo de unión es correcto respecto de un conjunto de precondiciones $P_1, P_2, \ldots, P_n$ y una postcondición Q si se verifica $P_i \implies Q \; \forall i \in \{1, \ldots, n\}$

- Al aplicar derivación directa, elegiremos $Q = P_1 \vee P_2 \vee \ldots P_n$
- Al aplicar derivación inversa, elegiremos $P_i = Q \; \forall i \in \{1, \ldots n\}$

Verificación de los nodos de unión



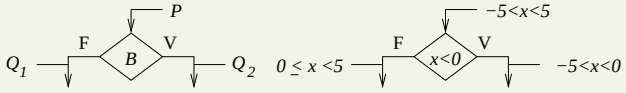
Nodo de decisión

Regla 2 Si una precondición P es válida antes de una decisión con condición B entonces para que el nodo sea correcto con respecto de P y Q_1, Q_2 debe ocurrir que:

- B sea evaluable.
- $P \wedge \neg B \Rightarrow Q_1$
- $P \wedge B \Rightarrow Q_2$

- Para la derivación directa, comprobaremos que B es evaluable y elegiremos $Q_1 = P \wedge \neg B$ y $Q_2 = P \wedge B$
- Para la derivación inversa elegiremos $P = (B \text{ evaluable}) \wedge ((Q_1 \wedge B) \vee (Q_2 \wedge \neg B))$

Nodo de decisión (II)

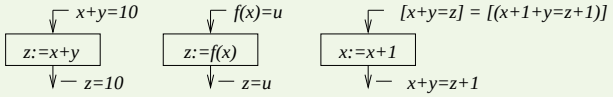


Nodo de asignación

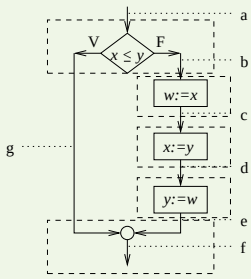
Regla 3 Si antes de una asignación $v := w$ es válida la precondición P, y la expresión w puede evaluarse, tras la asignación sigue siendo válido el mismo aserto sin más que substituir todas las apariciones de la expresión w por v.

- Para la derivación directa comprobamos que P garantiza que w se puede evaluar, escribimos P en función de la expresión w y sustituimos cada aparición de la expresión por v para obtener Q
- Para la derivación inversa sustituimos en Q cada aparición de v por la expresión w para construir P, añadiendo a este aserto la condición de que w sea evaluable

Nodo de asignación (II)



Un ejemplo de derivación directa

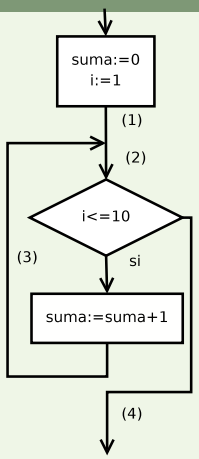


(a) $\{x = x_0, y = y_0, z = z_0\}$
(g) $\{x = x_0, y = y_0, z = z_0, x \leq y\}$
(b) $\{x = x_0, y = y_0, z = z_0, x > y\}$
(c) $\{w = x_0, x = x_0, y = y_0, z = z_0, w > y, x > y\}$
(d) $\{w = x_0, x = y_0, y = y_0, z = z_0, w > y$
(e) $\{w = x_0, x = y_0, y = x_0, z = z_0, y > x$
(f) $\{\{x_0, y_0\} = \{x, y\}, x \leq y, z = z_0\}$

Otros ejercicios simples

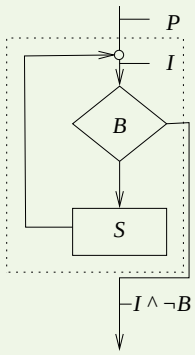
- Encontrar precondiciones que garanticen las postcondiciones indicadas:
  1. $\{\quad\}\ \ n := m\ \ \{n \leq 100\}$
  2. $\{\quad\}\ \ n := 2 \times n\ \ \{n \leq 100\}$
  3. $\{\quad\}\ \ n := n + 1\ \ \{n = n + 1\}$
  4. $\{\quad\}\ \ b := b \vee (k = 0)\ \ \{\neg b\}$
- Ahora encontrar postcondiciones garantizadas por las precondiciones indicadas:
  1. $\{n > 0\} \wedge \{z + u \times y = x \times y\}\ \ z := z + y; n := n + 1\ \ \{\quad\}$
  2. $\{r + q \times y = x\} \wedge \{r - y \geq 0\}\ \ r := r - y; q := q + 1\ \ \{\quad\}$
  3. $\{t < n\} \wedge \{x < 0\}\ \ x := x^2\ \ \{t < n\} \wedge \{\quad\}$

El problema de la iteración



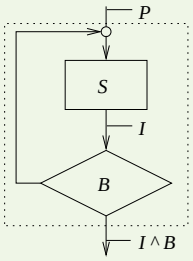
- Si deseamos verificar esta iteración simple
  - Partimos de la precondición
  - Para obtener (2) necesitamos (1) y (3)
  - Para obtener (3) necesitamos (2)
  - Tampoco sabemos si el bucle es finito
- Necesitamos otro esquema de razonamiento para demostrar sobre iteraciones
- Podemos aplicar *inducción* para determinar un valor para (2)

Corrección parcial



- Elegimos un *candidato a invariante* I justo antes de la condición de salida
- Nos aseguramos de que $P \implies I$
- Nos aseguramos de que $\{I \wedge B\}S\{I\}$ es parcialmente correcto
- Entonces si el bucle termina podemos garantizar $I \wedge \neg B$ como su postcondición

Corrección parcial (II)



- Elegimos un *candidato a invariante* I justo antes de la condición de salida
- Nos aseguramos de que $\{P\}S\{I\}$ es parcialmente correcto
- Nos aseguramos de que $\{I \wedge \neg B\}S\{I\}$ es parcialmente correcto
- Entonces si el bucle termina podemos garantizar $I \wedge B$ como su postcondición

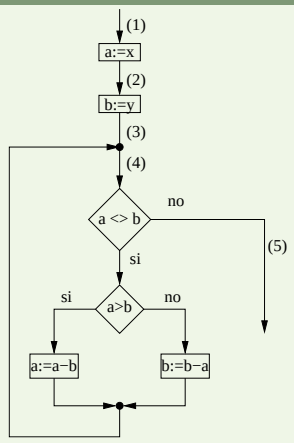
Búsqueda del candidato a invariante

- Podemos utilizar algunas estrategias que ayudan
  - Elaborar una traza del algoritmo
  - Intentar adivinar el objetivo de la iteración, expresarlo en lenguaje formal y retirar la condición de salida
- No es fácil elegir un buen candidato a invariante a posteriori
- Sería bueno que el programador nos hubiera dejado pistas

Finitud de los bucles

- Toda sucesión de enteros, estrictamente decreciente y acotada por el cero es finita
- Si construímos una sucesión en esos términos asociada a cada iteración del bucle, demostraremos que el bucle es finito
- También podemos calcular el número de iteraciones del bucle mediante el método de inducción

Planteamiento del problema



- Debemos llegar hasta la iteración aplicando las tres reglas de derivación

(1) $\{x, y \in \mathbb{Z}\} \wedge \{x, y > 0\}$

(2) $\{x, y \in \mathbb{Z}\} \wedge \{a = x > 0; y > 0\}$

(3) $\{x, y \in \mathbb{Z}\} \wedge \{a = x > 0; b = y > 0\}$

- Ahora necesitamos un candidato a invariante

(4) $\{\mathbf{mcd}(a, b) = \mathbf{mcd}(x, y)\} \wedge \{a, b > 0\}$

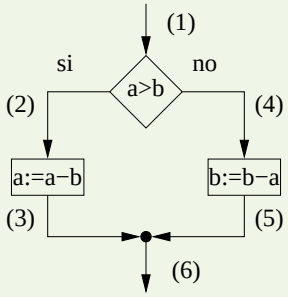
- El candidato puede ser útil porque:

  (3) $\Longrightarrow$ (4)

  $\{(4) \wedge a = b\} \Longrightarrow$

  $\Longrightarrow \{a = b = \mathbf{mcd}(x, y)\}$

Obtención del invariante



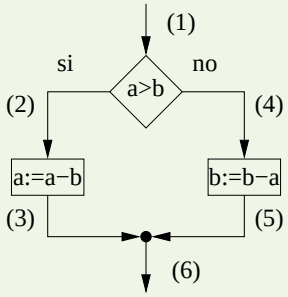
- Sólo falta ver el paso general de inducción porque *la precondición del bucle implica al candidato a invariante*

(1) $\{\mathbf{mcd}(a, b) = mcd(x, y)\} \wedge$
$\wedge \{a, b > 0\} \wedge \{a \neq b\}$

(2) $\{\mathbf{mcd}(a-b, b) = mcd(x, y)\} \wedge$
$\wedge \{a - b, b > 0\}$

(3) $\{\mathbf{mcd}(a, b) = mcd(x, y)\} \wedge$
$\wedge \{a, b > 0\}$

Obtención del invariante



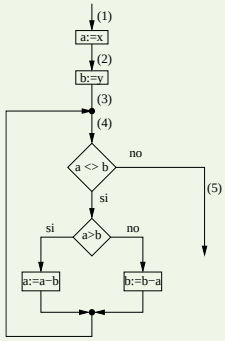
(4) $\{\mathbf{mcd}(a, b-a) = mcd(x, y)\} \wedge$
$\wedge \{a, b - a > 0\}$

(5) $\{\mathbf{mcd}(a, b) = mcd(x, y)\} \wedge$
$\wedge \{a, b > 0\}$

(6) $\{\mathbf{mcd}(a, b) = mcd(x, y)\} \wedge$
$\wedge \{a, b > 0\}$

- Luego el candidato a invariante lo es
- ¿Qué hemos demostrado?

Finitud de la iteración



- Necesitamos una sucesión de cota
- $c_k = \mathbf{max}(a_k, b_k)$ en el punto (4)

  $c_k > 0$ pues aplicando el invariante $a_k, b_k > 0$

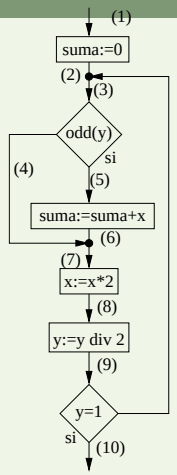
  $c_{k+1} = \mathbf{max}(a_{k+1}, b_{k+1})$

  $a_{k+1} = a_k - b_k$ **y** $b_{k+1} = b_k$ **o**

  $a_{k+1} = a_k$ **y** $b_{k+1} = b_k - a_k$

  En ambos casos $c_{k+1} < c_k$
- De modo que la sucesión y la iteración son finitas.
- La sucesión de cota garantiza finitud, pero no siempre indica número de ejecuciones de la iteración

Planteamiento del problema



- Debemos llegar hasta la iteración aplicando las tres reglas de derivación

(2) $\{x, y \in \mathbb{N}\} \wedge \{x = x_0,$
$y = y_0 > 1, suma = 0\}$

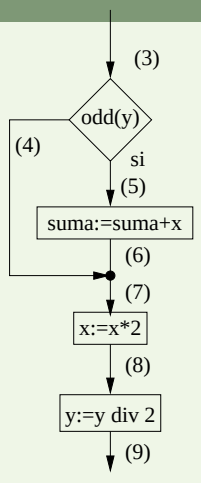
- Ahora necesitamos un candidato a invariante

(9) $\{xy + suma = x_0y_0\} \wedge \{y \geq 1\}$

- Podemos reducir la demostración porque:

  (2) $\Longrightarrow$ (9) $\wedge \{y \neq 1\}$

Obtención del invariante



- Sólo falta el paso general

(3) $\{xy + suma = x_0y_0\} \wedge \{y \geq 1\} \wedge \{y \neq 1\}$

(4) $\{xy + suma = x_0y_0\} \wedge \{y > 1\} \wedge \{y$ **par**$\}$

(5) $\{xy + suma = x_0y_0\} \wedge \{y > 1\} \wedge \{y$ **impar**$\}$

- Hay que expresar mejor (4) y (5)

(4) $\{x2y\mathbf{div}2 + suma = x_0y_0\} \wedge \{y > 1\}$

(5) $\{x2y\mathbf{div}2 + x + suma = x_0y_0\} \wedge \{y > 1\}$

(6) $\{x2y\mathbf{div}2 + suma = x_0y_0\} \wedge \{y > 1\}$

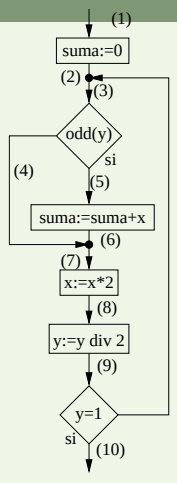
(7) $\{x2y\mathbf{div}2 + suma = x_0y_0\} \wedge \{y > 1\}$

(8) $\{xy\mathbf{div}2 + suma = x_0y_0\} \wedge \{y\mathbf{div}2 \geq 1\}$

(9) $\{xy + suma = x_0y_0\} \wedge \{y \geq 1\}$

- ¿Qué hemos demostrado?

Finitud de la iteración



- Necesitamos una sucesión de cota

  $c_k = y_k$ en el punto (9)

  $c_k > 0$ pues aplicando el invariante $y_k \geq 1$

  $c_{k+1} = y_k\mathbf{div}2 < y_k = c_k$
- De modo que la sucesión y la iteración son finitas.

Funciones Recursivas

- Si la solución del problema se define de forma inductiva, el resultado es un algorimo recursivo, que se llama a si mismo
- Tipos de recursividad
  - Directa / Indirecta
  - Lineal / Múltiple
  - Final / No final

Principio de Inducción

- $\left[P(0) \wedge \left(\forall n \in \mathbb{N},\ P(n) \Rightarrow P(n+1)\right)\right] \Rightarrow \forall n \in \mathbb{N}, P(n)$
- Para demostrar una propiedad P en $\mathbb{N}$ basta demostrar:
  - $P(0)$ es cierto (paso básico)
  - $\forall n \in \mathbb{N}\ P(n)$ (hipótesis de inducción) implica que $P(n+1)$ (Paso de la inducción)
- Del mismo modo se pueden definir funciones inductivas sobre $\mathbb{N}$:
  - Caso básico: $\mathbf{exp}(n, 0) = 1$
  - Caso general: $\mathbf{exp}(n, m+1) = \mathbf{exp}(n, m) \times n$

Definición Inductiva o Recursiva

- Siempre hay al menos un caso básico que se define de forma explícita
- La definición para un caso no explícito utiliza la definición de casos más simples
- Estamos suponiendo un orden en el dominio de los casos
  - $0! = 1\ \forall n \in \mathbb{N}, n > 0 n! = n \times (n-1)!$
  - $0!$ se define de forma explícita
  - $9!$ se define en función de un caso más simple ($9 \times 8!$)

Verificación de un algoritmo recursivo

```
algoritmo factorial (n: entero)
{ Precondición: n ∈ ℕ }
   inicio
      si n=0 entonces {Caso básico}
            devuelve 1
            {¿Postcondición?}
         si_no {¿Hipótesis de inducción?}
            devuelve n×factorial(n−1)
            {¿Postcondición?}
      fin_si
   fin
```


Verificación de un algoritmo recursivo

- Todas las demostraciones se hacen por inducción
- Corrección parcial:
  - Demostrar que es correcto en el o los casos base
  - Demostrar que si es correcto en un caso lo es en el siguiente
- Finitud
  - Demostrar que es finito en el o los casos base
  - Demostrar que si es finito en un caso lo es en el siguiente

Equipar el software con su especificación

- Dotando al software de especificación podemos...
  - Mostrar los motivos que nos hacen pensar que el software es correcto
  - Facilitar la comprensión del problema y su solución
  - Mejorar la autodocumentación del software
  - Fijar la base para la posterior depuración
  - Proteger, en cierta medida, el software de usos inadecuados
- Pero...
  - La especificación no debe formar parte de las estructuras de control que utilizamos
  - Los contratos separan lo *correcto* de lo *robusto*

Programación bajo contrato

- Sistema para equipar al software con sus especificaciones mediante *asertos*
- Los asertos declaran las condiciones de corrección como parte del programa y de su documentación
- Un aserto es una expresión booleana que puede cumplirse o no en determinado punto del programa
- Los asertos no tienen porqué ralentizar la ejecución del programa, puesto que pueden ser compilados o no. La decisión es tomada en el momento de compilación

La metáfora del contrato

- Dos partes implicadas: *Cliente* y *proveedor*
- Un *servicio*: Requerido por el cliente, proporcionado por el proveedor
- Dos tipos de asertos:
  - *Precondición:* Condiciones en que el proveedor puede prestar el servicio (`require`). Deben ser cumplidas por el cliente
  - *Postcondición:* Garantías que obtiene el cliente sobre la calidad del servicio (`ensure`). Deben ser cumplidas por el proveedor

El racional con su contrato

```
class RACIONAL
...
  inverso: RACIONAL is
      -- Número que multiplicado por Current produce el uno
    require
      no_nulo: not equal(Current, Current.cero)
    do
      create Result.make(denominador,numerador)
    ensure
      equal(Current.uno,Current*Result)
    end
  end; -- inverso
```


Consideraciones sobre los contratos

- Las clausulas `require` y `ensure` aparecen en la forma corta de la clase
- El incumplimiento del contrato «puede» activar una excepción (en función de las opciones de compilación)
- En principio ignoramos qué ocurrirá si si incumple el contrato,...
- pero podemos intentar controlarlo procesando las excepciones
- El cuerpo de una rutina *no comprobará en ningún caso* que se cumple su precondición
- El cliente *no comprobará en ningún caso* que se cumple la postcondición de su proveedor
- *Los asertos no son*:
  - Un sistema de chequeo de la entrada
  - Una nueva estructura de control

Consideraciones sobre los contratos (y II)

- Cada precondición de una rutina debe satisfacer los siguientes requisitos:
  - Debe ser justificable únicamente en términos de la especificación
  - Cada característica que aparece en la precondición del método debe estar exportada a todos los clientes potenciales del método
- En las postcondiciones podemos utilizar la construcción `old()` que hace referencia a la versión original de la entidad (Por ejemplo `ensure i= old(i)+1`)
- Disponemos de expresiones booleanas como `and then`, `or else` e `implies`

Imperativo vs. Aplicativo

`do; i:=i+1`	`ensure; i=old(i)+1`
Cómo	Qué
Operacional	Denotacional
Implementación	Especificación
Orden	Consulta
Imperativo	Aplicativo
Instrucción	Aserto

Invariantes de clase

- Asertos situados al final de la clase tras la cláusula `invariant`
- Son precondiciones para todos los métodos públicos de la clase
- Son postcondiciones para todos los métodos públicos de la clase
- Marcan las restricciones de *coherencia de la clase*
- Los *métodos de creación* tienen por objetivo llevar a un estado que satisface el invariante
- Todo método público que parte de un estado que satisface el invariante tiene la obligación de llevar el objeto a otro estado que satisface el invariante

Ejemplo de invariante

```
class RACIONAL
...
invariant
  denominador_no_nulo: denominador /= 0
  reducido1: denominador >0
  reducido2: numerador.abs.gcd(denominador)=1
  reducido3: numerador=0 implies denominador=1
end; -- class RACIONAL
```


Asertos en el código

```
x:=x^2+y^2
-- Código para despistar
check
  x>=0
  -- Es suma de cuadrados
end
y:=x.sqrt
```

- *Afirmamos* que se cumple un aserto
- Si el aserto no se cumple *puede*dispararse una excepción
- Requiere de un comentario explícito
- Facilita la comprensión del código
- Facilita la depuración del código

Invariante y variante de bucle

```
from
  -- Órdenes de inicialización
invariant
  -- Invariante del bucle
variant
  -- Sucesión de cota
until
  -- Condición de salida
loop
  -- Cuerpo de la iteración
end
```

- `invariant` y `variant` son optativos
- Si el aserto no se cumple *puede*dispararse una excepción
- Requiere de un comentario explícito
- Facilita la comprensión del código
- Facilita la depuración del código

Ejemplo de iteración

```
division(x,y: INTEGER): INTEGER is
  -- Cociente de la división entera de x entre y
  require
     parametros_positivos: x>0; y>0
  local
    q,r: INTEGER;
  do
    from r:= x
    invariant x=q*y+r; r>0
    variant r
    until r<y
    loop
      r:= r-y
      q:= q+1
    end -- loop
    Result:=q
  ensure
    resto_en_rango: r<y; r>0
    propiedad_cociente: x=q*y+r
  end
```


Otro ejemplo de iteración

```
mcd(x,y: INTEGER): INTEGER is
  -- Máximo común divisor de x e y
  require
   parametros_positivos: x>0; y>0
  local
    a,b: INTEGER;
  do
    from a:= x;b:= y
    invariant a>0; b>0; mcd(a,b)=mcd(x,y)
    variant a.max(b)
    until a=b
    loop
      if a>b then a:=a-b
        else b:=b-a
      end
    end
    Result:= a
  ensure -- ¿Sabemos si termina?
    result=mcd(x,y)
  end
```


Comentarios sobre los asertos

- Algunos asertos son difíciles o imposibles de escribir
  - Comentarios que explican el aserto que nos gustaría escribir
- Los asertos pueden contener llamadas a métodos, siempre que se trate de consultas que no modifiquen el objeto
- La ejecución de un método en la comprobación de un aserto no provoca comprobación de nuevos asertos
  - El peligro de la recursión infinita
  - ¿Quién vigila al vigilante?
- La mejor inspiración para construir un buen contrato es leer los contratos de buenas bibliotecas
- Existen opciones de compilación para activar, total o parcialmente, o incluso desactivar el chequeo de asertos (`compile -help`)