Introducción a la Algorítmica y Programación (3300)

Prof. Ariel Ferreira Szpiniak - aferreira@exa.unrc.edu.ar
Departamento de Computación
Facultad de Cs. Exactas, Fco-Qcas y Naturales
Universidad Nacional de Río Cuarto

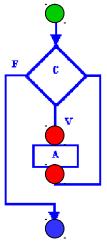
Teoría 13

Invariante de un ciclo

Recordando...
Composición Iterativa: Riesgos

La **coherencia** de las estructuras repetitivas depende de:

- Dado que las **acciones** involucradas en el cuerpo de la iteración pueden ser ejecutadas varias veces sucesivamente, la **poscondición** de las mismas debe ser **coherente** con la **precondición**.
- La **precondición** de la **estructura repetitiva** debe ser **coherente** con la **precondición** de las **acciones** involucradas en el cuerpo de la iteración.
- La **poscondición** de las **acciones** involucradas en el cuerpo de la iteración debe ser **coherente** con la **poscondición** de la **estructura repetitiva**.

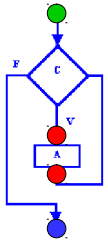


Composición Iterativa
Invariante

A los efectos de minimizar estos riesgos es importante tener en cuenta el concepto de **invariante** del ciclo.

El **invariante** de un ciclo es un **predicado lógico** que debe ser **verdadero** en ciertos momentos clave:

- antes de **ingresar** al ciclo
  dentro del **cuerpo** del **ciclo:**
  - **antes** de ejecutarse la **primer acción** del ciclo
  - **luego** de ejecutarse la **última acción** del ciclo.
- luego de **salir** del ciclo



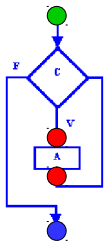
Todo ciclo posee algún invariante. Generalmente uno tiene la idea intuitiva de lo que debe hacer el ciclo. Eso es el invariante.

Composición Iterativa
Invariante

Ejemplo con mientras:

- {pred. invariante}
  mientras <condición de continuación> **hacer**
  - {pred. invariante y cond. de continuación}
    <acciones>
  - {pred. invariante}

  fmientras
- {pred. invariante y ¬cond. de continuación}



Composición Iterativa
Invariante

- **Este predicado invariante ayuda a desarrollar las acciones de la composición iterativa.**

- **El invariante debe cumplirse antes de iniciar el ciclo.**

- **El invariante debe cumplirse cada vez que se terminan de ejecutar las acciones correspondientes al *cuerpo* del ciclo.**
- **A la salida de un ciclo se debe cumplir el invariante y la condición de finalización del ciclo.**

Composición Iterativa
Invariante

- **El invariante es un predicado tal que si largáramos una corrida del programa y lo paráramos al inicio o al final del ciclo en cualquier iteración, su evaluación daría verdadera (si el ciclo está bien construido).**

- **El invariante puede expresarse en cualquier tipo de notación: informal, semi-formal o formal. Cuanto mayor sea el grado de formalización, más información aportará para razonar sobre las acciones involucradas en el cuerpo del ciclo y demostrar propiedades.**

Composición Iterativa
Invariante

Generalmente uno tiene la idea de lo que debe hacer el ciclo. Esa idea es la que se utiliza (consciente o inconscientemente) para determinar las acciones del ciclo. El invariante es entonces la especificación de lo que debe hacer el ciclo y las acciones son el cómo debe hacerse.

Sin embargo, no es una tarea sencilla determinar invariantes en un ciclo. De lo que sí debemos estar seguros es que si logramos encontrar un invariante seguramente habremos entendido lo que debemos hacer y nos será mucho más fácil encontrar las acciones. Además nos aportará cierto grado de corrección al ciclo, puesto que nos permitirá verificar que el ciclo cumpla con el objetivo perseguido.

Composición Iterativa
Invariante

¿Cómo se hace para encontrar invariantes?
Lo primero es saber con claridad que debo obtener luego del ciclo, es decir, debo saber cual es la poscondición del ciclo (no del problema, puesto no necesariamente en la misma).

Existen varias técnicas. Algunas de ellas son:

- **Reemplazar una constante por una variable en la poscondición del ciclo (una de las más usadas).**

- **Eliminar conjuntores de la poscondición del ciclo.**

- **Combinar la pre y la pos condición del ciclo.**

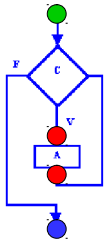
Composición Iterativa
Invariante

¿Cómo se puede aplicar el conocimiento del invariante de un ciclo al desarrollo de algoritmos?

- **Para verificar la correctitud del ciclo.**
- **Para derivar las acciones que deben escribirse antes, durante y una vez finalizado el ciclo.**

Ejemplo 1

Desarrollar un algoritmo que sume los primeros n números enteros, desde cero.



```
Algoritmo SumatoriaEnteros
Lexico
  n ∈ Z       //dato: límite superior de la sumatoria
  i ∈ Z       //auxiliar para la sumatoria parcial
  s ∈ Z       //resultado: sumatoria
Inicio
  Entrada:n
  s ← 0
  i ← 0
  mientras i < n hacer
    i ← i + 1
    s ← s + i
  fmientras
  // Sumatoria de los n primeros números
  Salida:s
Fin
```


Traza del ciclo

¿Cómo podemos ayudarnos para encontrar el invariante?

Una alternativa es determinar la traza del ciclo y tratar de encontrar en ella la relación existente entre las variables que intervienen en el ciclo.

La traza es el estado que toman las variables en cada iteración.

Por ejemplo, para el algoritmo anterior la traza es la siguiente:

Cantidad de iteraciones	i	s
0	0	0
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
...	...	...

Es decir que al finalizar el bloque de acciones, en cada iteración, la variable *s* contiene la sumatoria de los números que van desde *0* hasta *i*.

Por ejemplo, en la iteración nro. 6, *i* es igual a 6 y *s* es igual a 21, es decir, la suma de 0+1+2+3+4+5+6.

Por lo tanto el invariante sería:

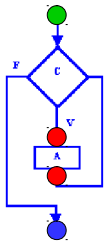
{Inv: s = sumatoria de 0 hasta i}

Ejemplo 1

```
Algoritmo SumatoriaEnteros
Lexico
  n ∈ Z       //dato: límite superior de la sumatoria
  i ∈ Z       //auxiliar para la sumatoria parcial
  s ∈ Z       //resultado: sumatoria
Inicio
  Entrada:n
  s ← 0
  i ← 0
  {Inv: s = sumatoria de 0 hasta 0 }
  mientras i < n hacer
    {Inv: s = sumatoria de 0 hasta i }
    i ← i + 1
    s ← s + i
    {Inv: s = sumatoria de 0 hasta i }
  fmientras
  {Inv: s = sumatoria de 0 hasta n} {Inv y ¬cond.continuación}
  Salida:s // Sumatoria de los n primeros números
Fin
```

¿Qué pasaría si fuera <=?

¿Qué pasaría si estuvieran invertidas?

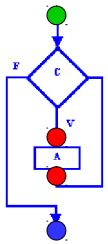


Evaluación del invariante

- Antes de entrar al ciclo

  s = sumatoria de 0 hasta i

  Como todavía no inicia el conteo i=0, luego s=0, esto implica que tanto i como s se deben inicializar en 0.



- Al comienzo y al final de las acciones de ciclo

  Supongamos que el número i es <=n y que estamos en la 3er iteración. En tal caso i=3 y s=1+2+3=6. Si esto se cumple es porque s está bien asignado (s+i) y que i está bien incrementado (i+1).

- Una vez que el ciclo concluyó

  s contiene la suma de los n primeros enteros. La condición de finalización del ciclo es i=n, en tal caso la evaluación del invariante debe ser s=sumatoria de los i números enteros y con i=n, s es la suma de los n primeros números.

Ejemplo 1

Veamos que sucedería si el invariante fuera distinto. Razonemos sobre el siguiente predicado invariante:

```
{Inv: s=sumatoria de 0 hasta i-1 }
```

i -1 es el entero acumulado en la suma hasta el punto en donde se evalúa el invariante.

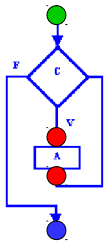
¿Cómo debería ser la traza?

Veamos el algoritmo:

Ejemplo 1

```
Algoritmo SumatoriaEnteros
Lexico
  n ∈ Z       //dato: límite superior de la sumatoria
  i ∈ Z       //auxiliar para la sumatoria parcial
  s ∈ Z       //resultado: sumatoria
Inicio
  Entrada:n
  s ← 0
  i ← 1
 {Inv: s = sumatoria de 0 hasta 0 }
  mientras i <= n hacer
   {Inv: s = sumatoria de 0 hasta i-1 }
    s ← s + i
    i ← i + 1
   {Inv: s = sumatoria de 0 hasta i-1 }
  fmientras
 {Inv: s = sumatoria de 0 hasta n+1-1}
  Salida:s // Sumatoria de los n primeros números
Fin
```

¿Qué pasaría si aquí fuera 0?



Uso del invariante

El invariante se puede utilizar de dos formas diferentes:

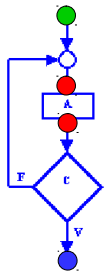
• A priori: para desarrollar las acciones que componen el bloque ciclo y encontrar la condición de continuación o terminación.

• A posteriori: para verificar que las acciones que componen el bloque ciclo son las correctas.

Ejemplo 2

Obtener ocho números enteros y luego informar cual es la suma de los ocho números.

```
Algoritmo Suma8Nros
Lexico
  n ∈ Z       //dato: números a sumar
  cant ∈ Z    //aux para el registro de la cantidad de números ingresados
  suma ∈ Z    //resultado: suma de los 8 números
Inicio
  cant ← 0
  suma ← 0

  repetir

    Entrada:n
    suma ← suma + n
    cant ← cant + 1

  hasta que cant=8

  Salida:suma // Suma de los 8 números
Fin
```



Traza del ciclo

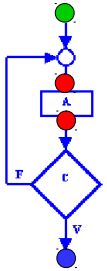
Cantidad de iteraciones	cant	suma	n
0			
1			
2			
3			
4			
5			
6			
7			
8			

{Inv: suma = sumatoria de los cant números ingresados}
O {Inv: suma = $(\sum_j: 0 \le j < cant: N_j)$ }

Ejemplo 2

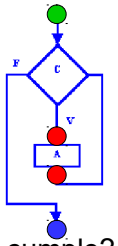
Desarrollar un algoritmo que permita a un usuario ingresar ocho números enteros y luego informe cual es la suma de los números ingresados.

```
Algoritmo Suma8Nros
Lexico
  n ∈ Z       //dato: números a sumar
  cant ∈ Z    //aux para el registro de la cantidad de números ingresados
  suma ∈ Z    //resultado: suma de los 8 números
Inicio
  cant ← 0
  suma ← 0
  {Inv: suma = (∑j: 0≤j<0: Nj) }
  repetir
    {Inv: suma = (∑j: 0≤j<cant: Nj) }
    Entrada:n
    suma ← suma + n
    cant ← cant + 1
    {Inv: suma = (∑j: 0≤j<cant: Nj) }
  hasta que cant=8
  {Inv: suma = (∑j: 0≤j<8: Nj) }
  Salida:suma //Suma de los 8 números
Fin
```



Evaluación del invariante

{Inv: suma = $(\sum_j: 0 \le j < cant: N_j)$ }



- Antes de entrar al ciclo

  No hay números ingresados. La cantidad de números es 0 (cant=0), suma es 0 y la sumatoria también por rango vacío (0=0).

- Al comienzo y al final de las acciones de ciclo

  La primera vez cant=1, un valor n es leído ($n=N_0$) y $suma=N_0$. ¿se cumple?

  La segunda vez cant=2, otro valor n es leído ($n=N_1$) y $suma=N_0+N_1$. ¿se cumple?

  Así sucesivamente ...

  Cuando cant=i, el valor n es leído ($n=N_{i-1}$) y $\mathbf{suma=(\sum_j: 0 \le j < cant: N_j)}$,es decir, $sum=N_0+...+N_{i-1}$. ¿se cumple?

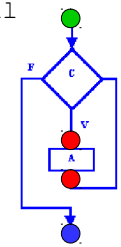
- Una vez que el ciclo concluyó

  cant=8, por lo tanto se leyeron 8 números $N_0,N_1,N_2,N_3,N_4,N_5,N_6,N_7$ y $\mathbf{suma=(\sum_j: 0 \le j < 8: N_j)}$,es decir, $suma=N_0+N_1+N_2+N_3+N_4+N_5+N_6+N_7$. ¿se cumple?

Ejemplo 3.1

Calcular el factorial de un número n (natural)

```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 0
  fact ← 1

  mientras i < n hacer

    i ← i + 1
    fact ← fact * i

  fmientras

  Salida:fact // Factorial de n
Fin
```



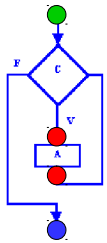
Traza del ciclo

Cantidad de iteraciones	i	fact
0		
1		
2		
3		
4		
5		
6		
7		
...	...	...

```
{Inv:                                                    }
```


Ejemplo 3.1

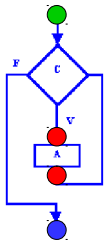
```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 0
  fact ← 1
● {Inv: fact = factorial de 0}
  mientras i < n hacer
  ● {Inv: fact = factorial de i}
    i ← i + 1
    fact ← fact * i
  ● {Inv: fact = factorial de i}
  fmientras
● {Inv: fact = factorial de n}

  Salida:fact // Factorial de n
Fin
```



Evaluación del invariante

`{Inv: fact = factorial de i }`

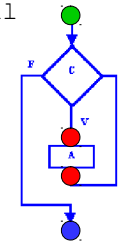
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 3.2

Otra forma.

```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 0
  fact ← 1

  mientras i < n hacer

    fact ← fact * (i+1)
    i ← i + 1

  fmientras

  Salida:facti // Factorial de n
Fin
```



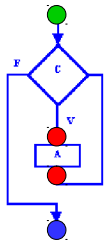
Traza del ciclo

Cantidad de iteraciones	i	fact
0		
1		
2		
3		
4		
5		
6		
7		
...	...	...

```
{Inv:                                                    }
```


Ejemplo 3.2

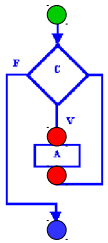
```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 0
  fact ← 1
{Inv: fact = factorial de 0}
  mientras i < n hacer
{Inv: fact = factorial de i}
    fact ← fact * (i+1)
    i ← i + 1
{Inv: fact = factorial de i}
  fmientras
{Inv: fact = factorial de n}
  Salida: fact // Factorial de n
Fin
```



Evaluación del invariante

```
{Inv: fact = factorial de i }
```

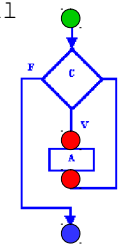
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 3.3

Otra forma.

```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 1
  fact ← 1

  mientras i <= n hacer

    fact ← fact * i
    i ← i + 1

  fmientras

  Salida:facti // Factorial de n
Fin
```



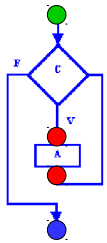
Traza del ciclo

Cantidad de iteraciones	i	fact
0		
1		
2		
3		
4		
5		
6		
7		
...	...	...

```
{Inv:                                                    }
```


Ejemplo 3.3

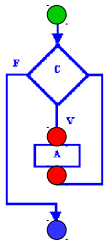
```
Algoritmo Factorial
Lexico
  n ∈ N       //dato: número ha calcular el factorial
  i ∈ Z       //auxiliar para calculo del factorial
  fact ∈ Z    //resultado: factorial de i
Inicio
  Entrada:n
  i ← 1
  fact ← 1
● {Inv: fact = factorial de 0}
  mientras i <= n hacer
  ● {Inv: fact = factorial de i-1}
    fact ← fact * i
    i ← i + 1
  ● {Inv: fact = factorial de i-1}
  fmientras
● {Inv: fact = factorial de n}
  Salida:fact // Factorial de n
Fin
```



Evaluación del invariante

```
{Inv: fact = factorial de i-1 }
```

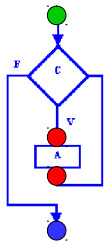
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 4

```
Algoritmo Sumatoria
Léxico
  M = 31
  TNros = arreglo [1..M] de Z
  j ∈ Z
  t ∈ Z
  compras ∈ TNros
  cargarCompras(resultado a ∈ TNros)
Inicio
  cargarcompras(compras)
  j ← 0
  t ← 0
  mientras j < M hacer
    j ← j + 1
    t ← t + (compras[j])
  fmientras
Fin
```

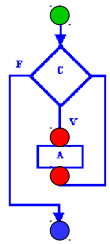
{Pos-cond: t = (∑i: 1≤i≤M: compras[i]) }



Evaluación del invariante

{Inv: t = (∑i: 1≤i≤j: compras[i]) }

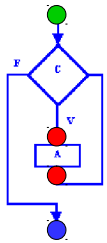
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 4.1

```
Algoritmo Sumatoria
Lexico
  M = 31
  TNros = arreglo [1..M] de Z
  j ∈ Z
  t ∈ Z
  compras ∈ TNros
  cargarCompras(resultado a ∈ TNros)
Inicio
  cargarcompras(compras)
  j ← 1
  t ← 0
  mientras j <= M hacer
    t ← t + (compras[j])
    j ← j + 1
  fmientras
Fin
```

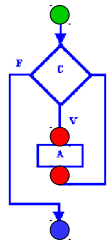
{Pos-cond: t = (∑i: 1≤i<M: compras[i]) }



Evaluación del invariante

{Inv: t = (∑i: 1≤i<j: compras[i]) }

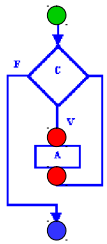
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 5

```
Algoritmo Sumatoria2
Lexico
  M = 31
  TNros = arreglo [1..M] de Z
  j ∈ Z
  t ∈ Z
  compras ∈ TNros
  cargarCompras(resultado a ∈ TNros)
Inicio
  cargarcompras(compras)
  j ← 0
  t ← 0
  mientras j < M hacer
    j ← j + 1
    si (j mod 2)=0
    entonces
        t ← t + (compras[j])
    fsi
  fmientras
Fin
```

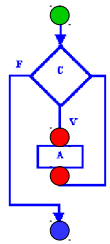
{Pos-cond: t = (∑i: 1≤i≤M ∧ (i mod 2)=0: compras[i]) }



Evaluación del invariante

{Inv: t = (∑i: 1≤i≤j ∧ (i mod 2)=0: compras[i]) }

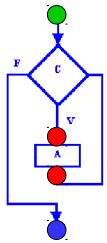
- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Ejemplo 5.1

```
Algoritmo Sumatoria2
Lexico
  M = 31
  TNros = arreglo [1..M] de Z
  j ∈ Z
  t ∈ Z
  compras ∈ TNros
  cargarCompras(resultado a ∈ TNros)
Inicio
  cargarcompras(compras)
  j ← 1
  t ← 0
  mientras j <= M hacer
    si (j mod 2)=0
    entonces
        t ← t + (compras[j])
    fsi
    j ← j + 1
  fmientras
Fin
```

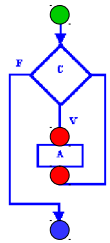
{Pos-cond: t = (∑i: 1≤i<M ∧ (i mod 2)=0: compras[i]) }



Evaluación del invariante

{Inv: t = (∑i: 1≤i<j ∧ (i mod 2)=0: compras[i]) }

- Antes de entrar al ciclo
  hacer...
- Al comienzo y al final de las acciones de ciclo
  hacer...
- Una vez que el ciclo concluyó
  hacer...



Bibliografía

– Ferreira Szpiniak, A.; Luna, C.D.; Medel, R. "Manual-Guía de Aprendizaje de Programación Avanzada". Editorial de la Fundación de la Universidad Nacional de Río Cuarto. Argentina. ISBN. 950-665-058-6. 1998. (Cap2, pags 73-90).

– Dijkstra, E.W. “A discipline of Programming”, Prentice Hall, Englewood Cliffs, 1976.

– Dijkstra, E.W.; Feijen, W.H.J. “A Method of Programming”, Addison-Wesley, 1988.

– Francez, N. “Program Verification”, Addison-Wesley, 1992.

– Gries, D. “The Science of Programming”, Springer-Verlag, 1981.

– Hoare, C.A.R.; Jones, C.B. (eds.) “Essays in Computing Science”, Prentice Hall,1989.

– Lucas, M.; Peyrin J-P.; Scholl P-C., “Algorítmica y Representación de Datos. Tomo 1”. Masson S. A., Paris. 1985.

– Myers, G.J. “El Arte de Probar el Software”, El Ateneo, 1984.

– Neumann, P.G.; et. al. “Risks to the Public”, Software Engineering Notes, Sección fija en Volúmenes 20 y 2, años 1995,1996.

– Sommerville, I. “Software Engineering”, Addison-Wesley, 1992.