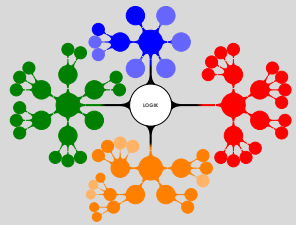
Tema 1
Sistemas Concurrentes y Distribuidos
Introducción

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 20 septiembre 2024

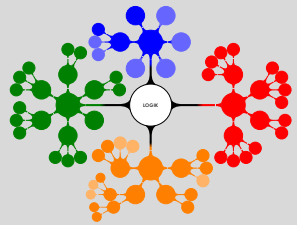
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



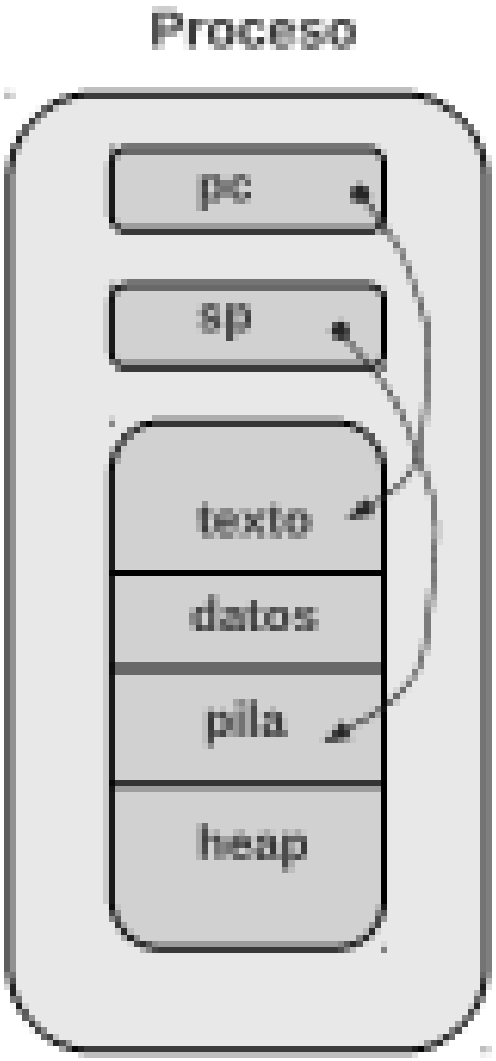
Concurrencia: programa y programación concurrente

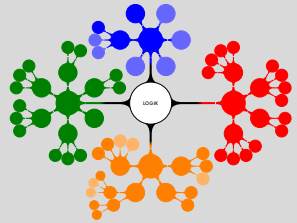
Programa Concurrente

1. Programa secuencial
   - Ejecución lineal de instrucciones.
2. Programa Concurrente
   - Múltiples unidades de ejecución independientes (llamadas procesos).
   - Los procesos cooperan para realizar tareas esenciales.
3. Proceso
   - Definición: Entidad de software abstracta, dinámica y activa.
   - Estado: No sólo las instrucciones; incluye su estado actual y la capacidad de interactuar con el medio ambiente.
4. Estado del proceso
   - Valores en registros (Contador de Programa - PC, Puntero de Pila - SP, Memoria Heap).
   - Acceso a recursos como archivos y dispositivos.
   - El estado debe estar protegido de otros procesos concurrentes.
5. Gestión de la concurrencia
   - Esencial para evitar el acceso incontrolado entre procesos.



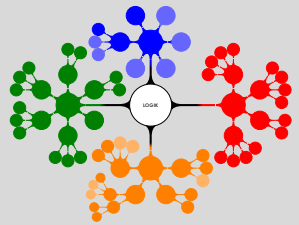
Proceso





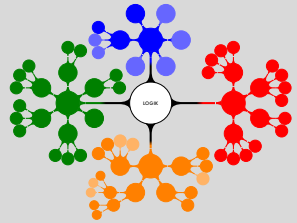
Conceptos básicos de procesos y concurrencia

1. Estructura de memoria del proceso
   - Dividido en zonas:
     - Instrucciones: Secuencia a ejecutar.
     - Área de datos: Para variables globales/estáticas.
     - Pila: Para variables locales y parámetros de procedimientos.
     - Memoria Heap: Variables dinámicas no estáticas.
2. Programas Concurrentes vs Secuenciales
   - Concurrencia: Los flujos de control se intercalan en menos núcleos.
   - Preserva el paralelismo lógico independientemente del número de núcleos.
   - Mejora la eficiencia al permitir la ejecución simultánea de múltiples hilos de control.
   - Reduce los retrasos en los cálculos causados por las operaciones de E/S.
3. Concurrencia en Simulación
   - Simula sistemas del mundo real de forma más natural.
   - Las actividades del mundo real pueden modelarse mejor con procesos concurrentes independientes.
4. Definición de concurrencia
   - El potencial de paralelismo en el código, independientemente de las limitaciones de hardware (número de núcleos/procesadores).



Key Concepts of Concurrent Programming

1. Modelo Abstracto de Computación
   - Expresa el paralelismo potencial a un alto nivel de abstracción.
   - Independiente de la arquitectura hardware del ordenador.
2. Concurrencia en la programación
   - Su objetivo es simplificar la sincronización y la comunicación entre los procesos.
   - Permite que el código se ejecute en diferentes arquitecturas.
   - Fomenta programas portables e independientes de la arquitectura.
3. Beneficios del modelo abstracto
   - Proporciona herramientas para diseñar y razonar sobre la concurrencia.
   - Simplifica el lenguaje de programación con primitivas de sincronización/comunicación de alto nivel.
   - Evita llamadas al sistema de bajo nivel o instrucciones específicas de la máquina.
4. Cinco axiomas de la programación concurrente
   - (i)Atomicidad e Intercalación de Instrucciones
   - (ii)Consistencia de datos después del acceso concurrente
   - (iii)Irrepetibilidad de las secuencias de instrucciones
   - (iv)Independencia de la velocidad de proceso

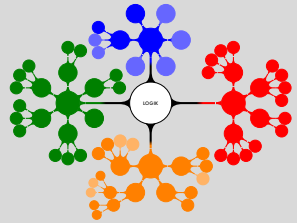


(i)Atomicidad e intercalación de instrucciones

Sentencia atómica(indivisible)

Una sentencia o instrucción de un proceso en un programa concurrente es atómica si siempre se ejecuta de principio a fin sin verse *afectada* (durante su ejecución) por las sentencias en ejecución de otros procesos del programa

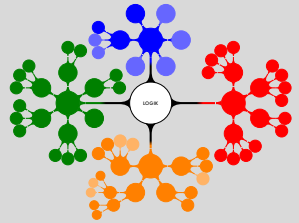
- No se verá afectada cuando el *funcionamiento* de dicha instrucción *no dependa nunca* de cómo se estén ejecutando otras instrucciones
- El funcionamiento de una instrucción se define por su efecto en el *estado de ejecución* del programa justo cuando ésta acaba
- El estado de ejecución de un programa concurrente está formado por los valores de las variables y de los registros de todos los procesos



Ejemplos de sentencias atómicas

Como ejemplo de instrucciones atómicas podemos citar muchas de las instrucciones máquina del repertorio de un procesador, por ejemplo las tres siguientes:

- Leer una celda de memoria (variable de tipo simple **x**) y cargar su valor en un registro (**r**) del procesador : `LOAD x`
- Incrementar el valor de un registro (**r**) u otras operaciones aritméticas sobre registros del procesador: `ADD r, a`
- Escribir el valor de un registro (**r**) en una celda de memoria: `STORE x`

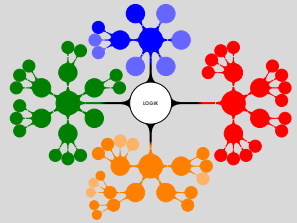


Sentencias atómicas: ejemplo

El resultado de estas instrucciones no dependerá nunca de otras instrucciones que se estén ejecutando concurrentemente.

Al finalizar, la celda de memoria o el registro (donde se escribe) tomará un valor concreto predecible siempre a partir del estado de inicio, es decir, el estado del proceso justo al terminar la ejecución de la sentencia está *determinado*.

- Si en un estado, el registro **r** tiene el valor *v* y se ejecuta la instrucción de escritura de **r** en una variable **x**, entonces en el estado final de **x** sabemos que vale *v*
- Lo anterior se puede afirmar incluso si existen otros procesos del programa accediendo a la variable **x**
- El estado del proceso, al terminar la ejecución de una sentencia atómica, estará bien definido y se puede representar como una función matemática que sólo depende del estado al inicio de dicha ejecución



Sentencias no atómicas

La mayoría de las sentencias de los lenguajes de programación de alto nivel son típicamente no atómicas, por ejemplo, la sentencia:

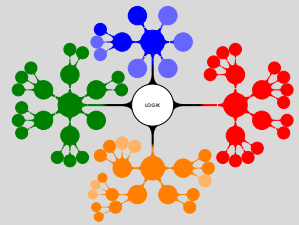
`x:= x + 1;` (incrementa el valor de la variable entera en 1 unidad)

El compilador utiliza una secuencia de tres sentencias:

1. Leer el valor de **x** y cargarlo en un registro **r** del procesador
2. Incrementar en una unidad el valor almacenado en el registro **r**
3. Escribir el valor del regsitro **r** en la variable **x**

El valor que toma la variable **x** justo al terminar la ejecución de la sentencia dependerá de que haya o no otras sentencias ejecutándose a la vez y tratando de escribir simultáneamente en la variable **x**

Decimos en este caso que existe *indeterminación*: no se puede predecir el estado final del proceso a partir de su estado inicial



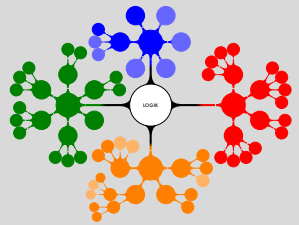
(i) Entrelazamiento de instrucciones atómicas

Programa concurrente C compuesto por 2 procesos secuenciales: P_A y P_B que se ejecutan a la vez

La ejecución de C puede dar lugar a cualquiera de los posibles entrelazamientos de sentencias atómicas de P_A y P_B:

Pr.	Posibles secuencias de instrucciones atómicas
P_A	$A_1\ A_2\ A_3\ A_4\ A_5$
P_B	$B_1\ B_2\ B_3\ B_4\ B_5$
C	$A_1\ A_2\ A_3\ A_4\ A_5\ B_1\ B_2\ B_3\ B_4\ B_5$
C	$B_1\ B_2\ B_3\ B_4\ B_5\ A_1\ A_2\ A_3\ A_4\ A_5$
C	$A_1\ B_1\ A_2\ B_2\ \ldots$
C	$B_1\ B_2\ A_1\ B_3\ B_4\ A_2\ \ldots$
C	$\ldots$

Las secuencia se va ordenando a medida que acaba cada una de las sentencias atómicas que la componen (que es cuando tienen efecto)



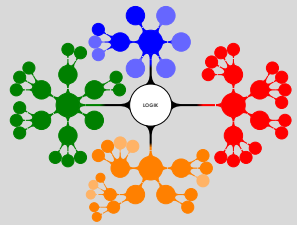
Modelo de programación basado en el entrelazamiento de instrucciones atómicas

El modelo basado en el estudio de todas las posibles secuencias de ejecución entrelazadas de los procesos de un programa concurrente constituye una **abstracción**:

- Se consideran sólo las *características relevantes* que determinan el resultado final del programa
- Nos permite simplificar el análisis y estudio de los programas

Se *ignoran detalles* de la computación *no relevantes* para obtener el resultado, como por ejemplo:

- El estado de la memoria asignado a cada proceso
- Los registros particulares a los que accederá cada proceso
- El costo de los cambios de contexto entre procesos que realiza el sistema operativo
- La política concreta de planificación de los procesos
- Diferencias de velocidad entre entornos multiprocesador y monoprocesador



(ii) Consistencia de los datos tras el acceso simultáneo

El entrelazamiento de instrucciones atómicas preserva la consistencia de los resultados

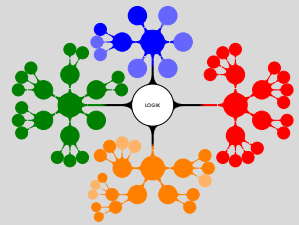
El resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución

Independencia total del entorno de ejecución de los procesos o del programa

Suponemos un programa P que se compone de 2 instrucciones atómicas, I_0 e I_1, que se ejecutan concurrentemente, es decir, $P \equiv I_0 \| I_1$, entonces:

- Si I_0 e I_1 no acceden a la misma celda de memoria o registro, entonces el orden de ejecución de dichas instrucciones no afectará al resultado final
- Si no (acceden a la misma celda M): Si $I_0 \equiv M \leftarrow 1$ e $I_1 \equiv M \leftarrow 2$, la única suposición razonable es que el resultado sea consistente.
- Por tanto, al final $M = 1$ ó $M = 2$, pero nunca podría alcanzar otro valor, por ejemplo: $M = 3$

En caso de que no se cumpliera, sería imposible poder razonar sobre las propiedades de corrección de los programas concurrentes



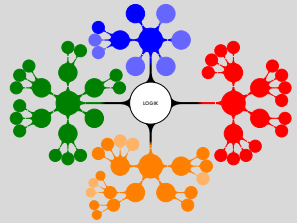
(iii) Irrepetibilidad de las secuencias de instrucciones

Estado de un programa concurrente

- Valores de las variables del programa en un momento dado. Incluyen variables declaradas explícitamente y variables con información de estado oculta (contador del programa, registros, ...)
- Un programa concurrente comienza su ejecución en un estado inicial y los procesos van modificando el estado conforme se van ejecutando sus sentencias atómicas (producen transiciones entre dos estados de forma indivisible)

Historia o traza de un programa concurrente: Secuencia de estados $s_0 \rightarrow s_1 \rightarrow \ \ldots \rightarrow s_n$, producida por una secuencia concreta de entrelazamiento

Irrepetiblidad de las historias: La probabilidad de que 2 trazas de ejecución seguidas del mismo programa coincidan es prácticamente "0"



(iv) Independencia de la velocidad del proceso

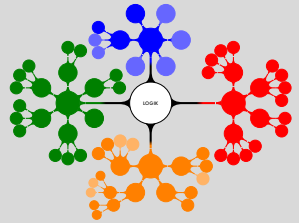
No se puede hacer ninguna suposición acerca de las velocidades absolutas/relativas de ejecución de los procesos, salvo que es mayor que cero

Un programa concurrente se entiende sólo en base a sus componentes (procesos) y sus interacciones, sin tener en cuenta el entorno de ejecución.

Ejemplo: un disco es más lento que el procesador, pero no se debe asumir este hecho en el diseño de un programa concurrente.

Si se hicieran suposiciones que dependiesen del tiempo o de la velocidad de ejecución de los procesos:

- Sería difícil detectar y corregir fallos
- La corrección dependería de la configuración de ejecución, que puede cambiar



(v) Hipótesis del progreso finito

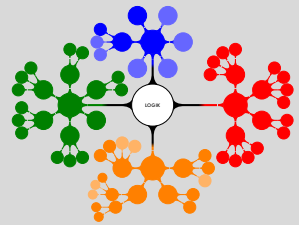
Si se cumple la hipótesis, la velocidad de ejecución de cada proceso será no nula, lo cual tiene estas dos consecuencias:

Punto de vista global

- Durante la ejecución de un programa concurrente, en cualquier momento existirá al menos 1 proceso preparado, es decir, eventualmente el planificador de bajo nivel del sistema seleccionará a algún proceso para su ejecución

Punto de vista local

- Cuando un proceso concreto de un programa concurrente comienza la ejecución de una sentencia, completará la ejecución de la sentencia en un intervalo de tiempo finito



Notaciones para expresar ejecución concurrente

Propuestas iniciales: no separan la definición de los procesos de su sincronización

Propuestas posteriores: separan ambos conceptos e imponen una estructura al programa concurrente, diferente de uno secuencial

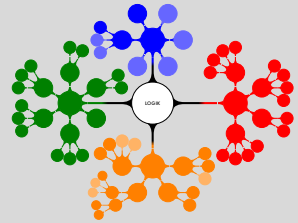
Declaración de procesos: rutinas específicas de programación concurrente $\Rightarrow$ Estructura del programa concurrente más clara

Sistemas Estáticos:

- Número de procesos fijado en el fuente del programa
- Los procesos se activan al lanzar el programa
- Ejemplo: Message Passing Interface (MPI-1)

Sistemas Dinámicos:

- Número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución
- Ejemplos: OpenMP, PThreads, Java Threads, MPI-2

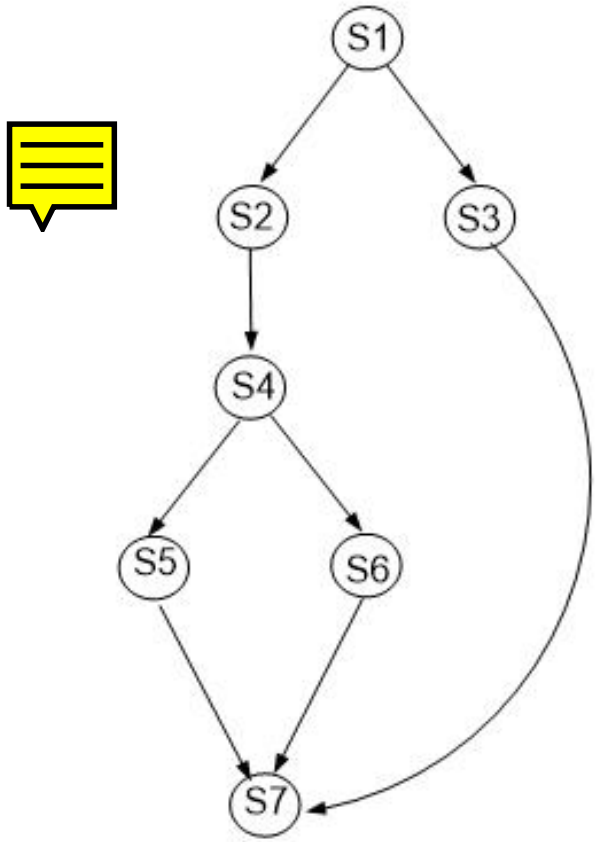


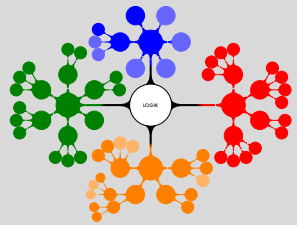
Grafo de sincronización

El Grafo de Sincronización es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (actividad):

Dadas dos actividades, S_1 y S_2, una arista (flecha) desde S_1 hacia S_2 significa que S_2 no puede comenzar su ejecución hasta que S_2 haya finalizado

Muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa





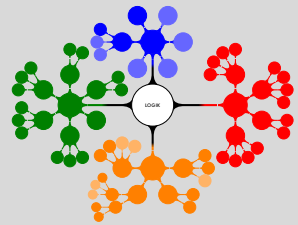
Definición estática de procesos

```
var ... \\variables compartidas

process Uno;
var ... \\variables locales
begin
  ... \\codigo
end;
process Dos;
var ... \\variables locales
begin
  ... \\codigo
end;
... \\otros procesos
```

Se *lanza* la ejecución concurrente de ambos procesos en 1 bloque del programa:

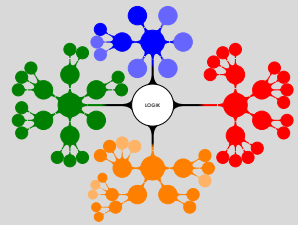
```
cobegin  Uno || Dos coend;
```

El programa acaba cuando acaban de ejecutarse todas las instrucciones de 2 los procesos. Las variables compartidas se inicializan antes de comenzar la ejecución concurrente de los procesos



Definición estática de vectores de procesos

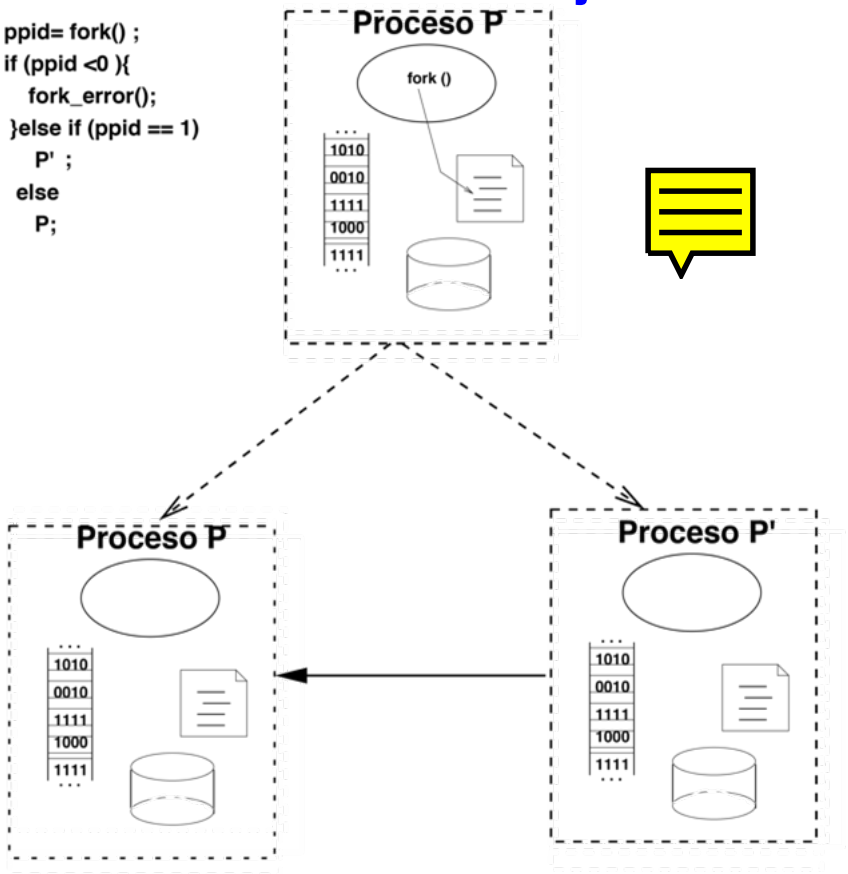
```
var ... \\variables compartidas

process NomP[ind : a.. b];
var ... \\variables locales
begin
  ... \\codigo
  ...\\ (ind vale a, a+1, ... b)
end;
... \\otros procesos
```

Creación de procesos no estructurada: fork-join

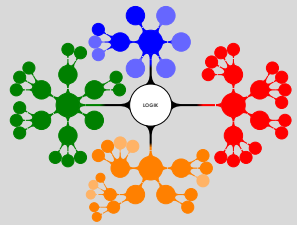
```
process P2;
 begin
  D;
 end;
process P1;
 begin
  A;
  fork P2;
  B;
  join P2;
  C;
 end;
```

```
ppid= fork() ;
if (ppid <0 ){
   fork_error();
}else if (ppid == 1)
   P' ;
 else
   P;
```



fork: sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente (bifurcación)

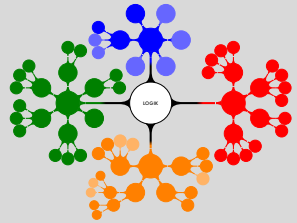
join: sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente (unión)



Introducción *exclusión mutua* y *sincronización*

No todas las secuencias de entrelazamiento de las instrucciones de los procesos a que da lugar la ejecución de un programa concurrente son posibles

- Los procesos no suelen ejecutarse de una forma totalmente independiente, sino que que **colaboran** entre ellos
- **Condición de sincronización:** restricción en el orden en que se pueden entremezclar las instrucciones que generan los procesos de un programa
- **Exclusión mutua:** se da en secuencias finitas de intrucciones del código de un programa, que han de ejecutarse de principio a fin por un único proceso, que no puede ser desplazado del procesador mientras ejecuta esta sección crítica de instrucciones

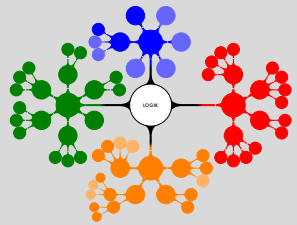


Instrucciones compuestas e instrucciones atómicas

En pseudo-código, podemos convertir las sentencias compuestas a atómicas: indicando que se deben de ejecutar sin ser interrumpidas, usando los caracteres $<$ y $>$:

```
{ instr. compuestas (no atómicas) }
begin
x := 0 ;
cobegin
x:= x+1 ;
x:= x-1 ;
coend
end
////////////////////////////
{ instr. atómicas }
begin
x := 0 ;
cobegin
< x:= x+1 > ;
< x:= x-1 > ;
coend
end
```

- En el primer código, al acabar, **x** puede tener un valor cualquiera del conjunto $\{-1, 0, 1\}$
- En el segundo código **x** finaliza siempre con el valor 0

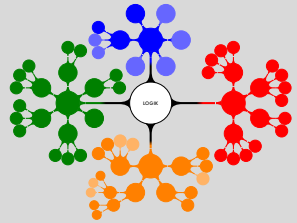


Condición de sincronización

En un programa concurrente, una condición de sincronización establece para asegurarnos de que todas las trazas del programa son correctas

- Suele ocurrir cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos)

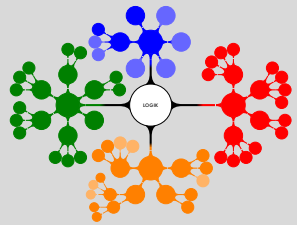
Un ejemplo sencillo de **condición de sincronización**: en el ejemplo del productor-consumidor, que los productores esperen a que el búfer no esté lleno.



El paradigma del *Productor Consumidor*

Dos procesos cooperantes en los cuales uno de ellos (*productor*) genera una secuencia de valores (p.ej.enteros) y el otro (*consumidor*) utiliza cada uno de estos valores

```
var x : integer ; { contiene cada valor producido }
{ Proceso productor: calcula 'x' }
process Productor ;
var a : integer ; { no compartida }
begin
 while true do begin
 { calcular un valor }
  a := ProducirValor() ;
  { escribir en mem. compartida }
  x := a ; { sentencia E }
 end
end
process Consumidor ; { Proceso Consumidor: lee 'x' }
var b : integer ; { no compartida }
begin
 while true do begin
 { leer de mem. compartida }
  b := x ; { sentencia L }
  { utilizar el valor leido }
  UsarValor(b) ;
 end
end
```



Trazas incorrectas de un programa concurrente

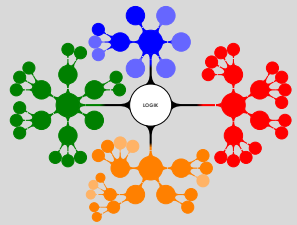
Los procesos sólo *funcionan* como se espera si el orden en el que se entrelazan las sentencias elementales etiquetadas como E (escritura) y L (lectura) es: E, L, E, L, E, L, ...

Trazas incorrecetas del programa *Productor Consumidor*:

- **L, E, L, E,** . . . es incorrecta: se hace una lectura de **x** previa a cualquier escritura (se lee valor indeterminado)
- **E, L, E, E, L,** . . . es incorrecta: hay dos escrituras sin ninguna lectura entre ellas (se produce un valor que no se lee)
- **E, L, L, E, L,** . . . es incorrecta (para el código del P/C anterior): hay dos lecturas de un mismo valor que, por tanto, es usado dos veces

La secuencia válida asegura la condición de sincronización:

- *Consumidor* no lee hasta que *Productor* escribe un nuevo valor en x (cada valor producido es usado una sola vez)
- *Productor* no escribe un nuevo valor hasta que *Consumidor* lea el último valor almacenado en **x** (ningún valor producido se pierde)

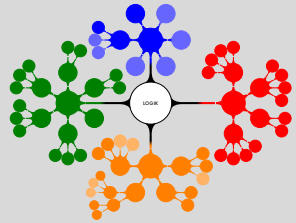


Concepto de corrección de un programa concurrente

Propiedad de un programa concurrente: Algo que se puede afirmar del programa que es cierto para todas las posibles trazas del programa

Hay 2 tipos de propiedades:

- Propiedad de seguridad (safety).
- Propiedad de vivacidad (liveness).



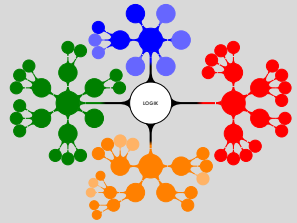
Propiedad de *seguridad*

Condiciones que deben cumplirse en cada instante de la traza del programa; son del tipo: *nunca pasará nada malo*

- Requeridas en especificaciones estáticas del programa
- Son fáciles de demostrar y para cumplirlas se suelen impedir determinadas posibles trazas

Ejemplos:

- *Exclusión mutua*: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones
- *Ausencia Interbloqueo* (Deadlock-freedom): Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá
- *Propiedad de seguridad en el Productor-Consumidor*: El consumidor debe consumir todos los datos producidos por el productor en el orden en que se van produciendo



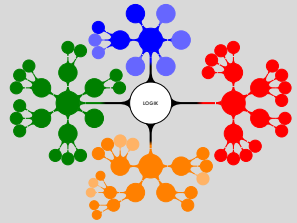
Propiedades de *vivacidad*

Son propiedades que deben cumplirse en algún momento futuro y no especificado del programa; son del tipo: *realmente sucede (o va a suceder) algo bueno*

- Son propiedades dinámicas, más difíciles de demostrar que las propiedades de seguridad

Ejemplos:

- *Ausencia de inanición* (starvation-freedom): Un proceso o grupo de procesos del programa no puede ser indefinidamente pospuesto. Asegura que, en algún momento, podrá avanzar
- *Equidad* (fairness): Tipo particular de propiedad de vivacidad. Un proceso que pueda progresar debe hacerlo con justicia relativa con respecto a los demás procesos del programa. Más ligado a la implementación y a veces se incumple: existen distintos grados



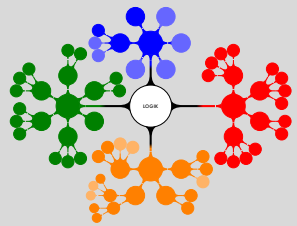
Pruebas simples de programas concurrentes

¿Cómo demostrar que un programa cumple una determinada propiedad ?

- *Posibilidad*: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad
- *Problema*: Sólo permite considerar un número finito y muy limitado de trazas del programa y no demuestra la ausencia de casos indeseables (en alguna traza no explorada)
- Ejemplo: Comprobar que el proceso P produce al final `x == 3`:

```
process P ;
var x : integer := 0 ;
cobegin
x = x+1 ; x = x+2 ;
coend
```

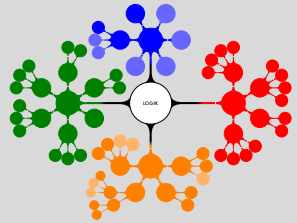
(hay varias trazas que llevan al resultado: x==1 o x==2, y estas historias podrían ocurrir en algunas ejecuciones, por tanto, produciendo un resultado no-determinado)



Enfoque operacional: análisis exhaustivo

- *Enfoque operacional*: Análisis exhaustivo de casos. Se comprueba la corrección de todas las posibles trazas
- *Problema*: Su utilidad está muy limitada cuando se aplica a los programas concurrentes, ya que el número de entrelazamientos crece exponencialmente con el número de instrucciones de los procesos

Para el sencillo programa P, formado por 2 procesos y 3 sentencias atómicas por proceso, tendríamos que estudiar 20 historias diferentes (=*permutaciones con repetición de 6 elementos con 2 grupos de 3 elementos*)



Verificación: enfoque axiomático

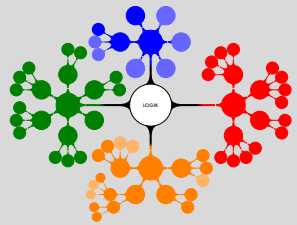
Se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia

- Se usan fórmulas lógicas (llamados: *asertos* o *predicados*) para caracterizar un conjunto de estados
- Las sentencias atómicas actúan como transformadores de tales asertos. Los teoremas se escriben de la forma:

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

Interpretación de un teorema (llamado también: *triple*) en esta lógica: "*Si la ejecución de la sentencia* **S** *comienza en algún estado en el que es verdadero el aserto* **P** *(precondición), entonces el aserto* **Q** *(poscondición) será verdadero en el estado resultante*"

Menor Complejidad que el enfoque operacional: El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa

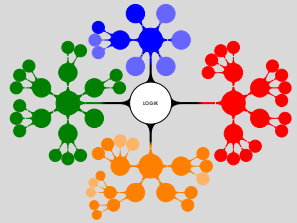


Notación: *Triples* de Hoare

- Se introduce la notación de *especificación de corrección parcial* para especificar lo que hace un programa

$$\{P\}\ C\ \{Q\}$$

  - C es un programa del lenguaje cuyos programas están siendo especificados
  - P y Q son asertos definidos con las variables programa que representa C
  - Las variables libres de P y Q son del programa o son variables lógicas
- Los asertos caracterizan los estados aceptables del programa:
  - V caracteriza a todos los estados del programa; sin embargo, F no caracteriza a ninguno
  - Los *triples* ({P} C {Q}) son los teoremas (= *proposiciones demostrables*) con la lógica que estamos tratando



Significado de la notación de Hoare

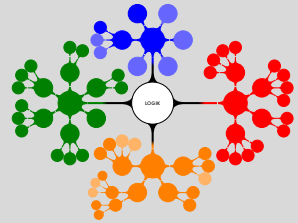
{P} *C* {Q} es cierto si

- siempre que *C* es ejecutado en un estado que
- satisface P y *si* la ejecución de *C* termina,
- el estado en que *C* termina satisface Q

Ejemplo: $\{X == 1\}\ X = X + 1\ \{X == 2\}$

- el aserto P dice que el valor de X es 1
- el aserto Q dice que el valor de X es 2
- C es la sentencia de asignación $X = X + 1$

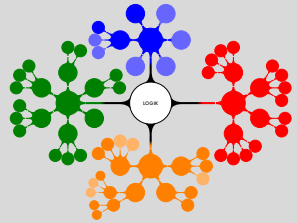
- $\{X == 1\}\ X = X + 1\ \{X == 2\}$ es cierto
- $\{X == 1\}\ X = X + 1\ \{X == 3\}$ es falso
- $\{X == 1\}$ *WHILE T DO NULL* $\{Y == 3\}$ ¡es cierto!



La estructura de las demostraciones

Una demostración es una secuencia de líneas (o *triples*) de la forma {P} *C* {Q}, cada una de los cuales es un *axioma* o deriva de los anteriores aplicando una *regla de inferencia* de la lógica

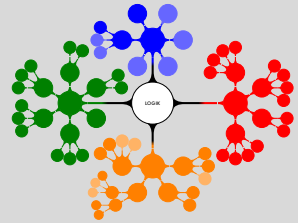
- Una demostración consiste en una secuencia de *líneas*:
  - como la definición de $()^2$ del ejemplo siguiente
- Cada una de las líneas es una instancia de un *axioma*
- o se deriva de las líneas anteriores por medio de una *regla de inferencia*
- La sentencia de la última línea de una demostración es lo que se quiere demostrar
  - como ocurre con: $(\mathtt{X}+1)^2 == \mathtt{X}^2 + 2 \times \mathtt{X} + 1$



Ejemplo de demostración

- Ejemplo de una demostración formal:

1. $(X+1)^2 == (X+1) \times (X+1)$
   Definición de $()^2$
2. $(X+1) \times (X+1) == (X+1) \times X + (X+1) \times 1$
   Distributiva izquierda de $\times$ con $+$
3. $(X+1)^2 == (X+1) \times X + (X+1) \times 1$
   Sustituir línea 2 en 1
4. $(X+1) \times 1 == X+1$ Ley identidad para 1
5. $(X+1) \times X == X \times X + 1 \times X$
   Distributiva derecha de $\times$ con $+$
6. $(X+1)^2 == X \times X + 1 \times X + X + 1$
   Sustituir líneas 4 y 5 en 3
7. $1 \times X == X$ Ley identidad para 1
8. $(X+1)^2 == X \times X + X + X + 1$
   Sustituir línea 7 en 6
9. $X \times X == X^2$ Definición de $()^2$
10. $X + X == 2 \times X$ 2==1+1, ley distributiva
11. $(X+1)^2 == X^2 + 2 \times X + 1$
    Sustituir líneas 9 y 10 en 8

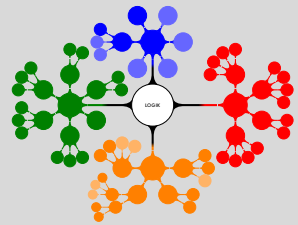


Propiedades de seguridad y complección

- Un aserto {P} caracteriza un estado aceptable del programa, es decir, un estado que podría ser alcanzado por el programa si sus variables tomaran unos determinados valores
- Concepto de Interpretación de las fórmulas de un SLF:

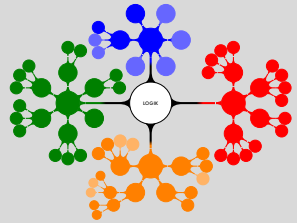
$$\text{asertos} \rightarrow \{V, F\}$$

  - Fórmula satisfascible
  - Fórmula válida
  - el aserto { V } caracteriza a todos los estados del programa (o *precondición más débil*)
  - el aserto { F } no se cumple por parte de ningún estado del programa (o *precondición más fuerte*)
- *SLF* seguro: $\text{asertos} \subseteq \{\text{hechos ciertos}\}$ – expresados como fórmulas de la lógica
- *SLF* completo: $\{\text{hechos ciertos}\} \subseteq \{\text{asertos}\}$
- Los sistemas lógicos para demostración de programas no suelen poseer la propiedad de complección, pero a efectos prácticos es suficiente con la de complección relativa (todas las proposiciones son demostrables excepto si contienen expresiones aritméticas)



Fórmulas proposicionales válidas (tautologías)

- Leyes distributivas:
  - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
  - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
- Leyes de De Morgan:
  - $\neg(P \wedge Q) = \neg P \vee \neg Q$
  - $\neg(P \vee Q) = \neg P \wedge \neg Q$
- Eliminación-*And*: $(P \wedge Q) \rightarrow P$
- Eliminación-*Or*: $P \rightarrow (P \vee Q)$

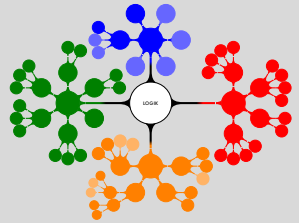


Axiomas y reglas de inferencia

Sirven para poder llevar a cabo las demostraciones de programas, ya que cada línea de la demostración es o bien un axioma o se deriva de la anterior mediante una regla de inferencia

- Axioma: fórmulas que sabemos que son ciertas en cualquier estado del programa
- Regla de inferencia: para derivar fórmulas ciertas a partir de los axiomas o de otras que se han demostrado ciertos previamente

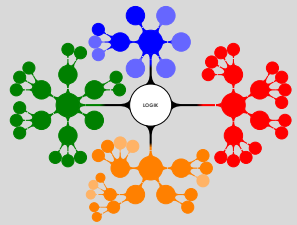
Notación de una Regla de inferencia:
(*nombre de la regla*) $\frac{H_1, H_2, \ldots, H_n}{C}$
Un aserto (o *teorema*) es una fórmula con una interpretación cierta que pertenece al dominio de los hechos que queremos probar (o *dominio del discurso*)
Los asertos de nuestra lógica coinciden con las líneas o sentencias lógicas de las que se compone la demostración de un programa



Axiomas y reglas de inferencia–II

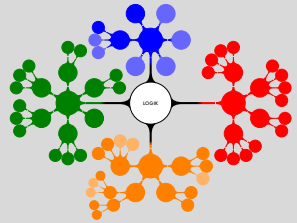
1. Axioma de la sentencia nula: { P } NULL { P}: si el aserto es cierto antes de ejecutarse esta sentencia, tendrá la misma interpretación cuando la sentencia termine
2. Sustitución textual: $\{P_e^x\}$ es el resultado de sustituir la expresión *e* en cualquier aparición de la variable *x* en P
3. Axioma de asignación: $\{P_e^x\}x = e\{P\}$: una asignación cambia solo el valor de la *variable objetivo*, el resto de las variables conservan los mismos valores.
   Ejemplos:
   A. $\{V\}\, x = 5\, \{x == 5\} \equiv \{P\}$ es un *aserto* pues:
      $P_5^x \equiv \{x == 5\}_5^x \equiv \{5 == 5\} \equiv V$
   B. $\{x > 0\}\, x = x + 1\, \{x > 1\} \equiv \{P\}$ es un aserto pues:
      $P_{x+1}^x \equiv \{x > 1\}_{x+1}^x \equiv \{x + 1 > 1\} \equiv \{x > 0\}$

   Reglas para *conectar* los triples en las demostraciones:
4. Regla de la consecuencia (1): $\dfrac{\{P\}S\{Q\},\ \{Q\} \rightarrow \{R\}}{\{P\}S\{R\}}$



Axiomas y reglas de inferencia–II

1. Regla de la consecuencia (2): $\dfrac{\{R\} \rightarrow \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}}$
2. Regla de la composición: $\dfrac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}}$: para obtener la pre y pos-condición de 2 sentencias juntas
3. Regla del IF: $\dfrac{\{P\} \wedge \{B\}S_1\{Q\},\ \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\mathit{if}\ B\ \mathit{then}\ S_1\ \ \mathit{else}\ S_2\ \mathit{fi}\{Q\}}$
4. Regla de la iteración: $\dfrac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\mathit{while}\ B\ \mathit{do}\ S\ \ \mathit{enddo}\{I \wedge \neg B\}}$: podrá iterar un número arbitrario de veces (incluso 0); el *invariante* I se satisface antes y después de cada iteración



Verificación de sentencias concurrentes

- La ejecución concurrente de los procesos hace que el SLF deje de ser seguro: problema de la *interferencia*

```
y=0; z=0;
cobegin
 x= y+z || y=1; z=2
coend;
```

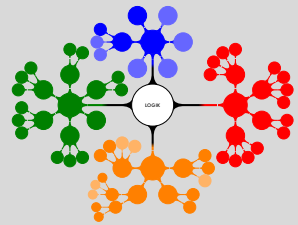
estados del programa	valores de x
x==y==z==0	0
x==y==1, z==0	1
x==3, y==1, z==2	3
-	2

¿Cómo evitar la interferencia?

- Evaluación de expresiones que no hacen referencia a variables modificadas concurrentemente

```
 {x==0; y==0}
  x= 0; y=0;
  cobegin x= x+1 || y= y+1 coend;
  z=x+y
{x==1, y==1, z==2}
```

- ¡Condición demasiado restrictiva!: casi ningún programa la cumpliría.



Verificación de sentencias concurrentes-II

Acción atómica elemental $< \ldots >$

Propiedad como máximo una vez: " *La evaluación de una expresión por un proceso concurrente se hace de forma atómica si las variables que comparte son leídas o escritas por un único proceso*"

Ejemplo 1: *Evaluación de expresiones*

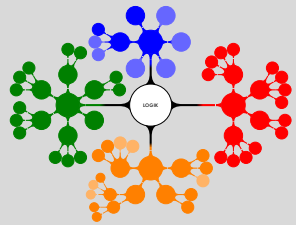
1. $x = 0; y = 0;$ *cobegin* $x = y + 1 || y = x + 1$ *coend*; ¡Incorrecta!
2. $x = 0; y = 0; z = Z;$ *cobegin* $x = y + 1 || y = z + 1$ *coend*;
3. $x = 0; y = 0; z = Z;$ *cobegin* $x = z + 1 || y = x + 1$ *coend*;

No interferen. entre un aserto y una instrucción atómica *a*:

$$\{\mathrm{C} \wedge \mathrm{pre}(\mathrm{a})\}\,\mathrm{a}\,\{\mathrm{C}\}$$

Ejemplo de no-interferencia:

1. $\{y = 0, z = Z\} < y = z + 1 >$ no interfiere con la poscondición de $< x = y + 1 > \{x = Y + 1, z = Z\}$
2. $\{x = 0, z = Z\} < x = y + 1 >$ no interfiere con la poscondición de $< y = z + 1 > \{y = Z + 1, z = Z\}$



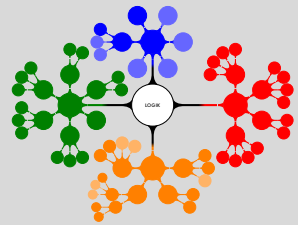
Regla de inferencia de la composición concurrente de sentencias

Si $\{P_i\}\, S_i\, \{Q_i\}$ no interfieren, entonces se puede demostrar el triple:

$$\{P_1 \wedge P_2 \ldots \wedge P_n\}$$
$$\text{COBEGIN}$$
$$S_1 \| S_2 \| \ldots \| S_n$$
$$\text{COEND}$$
$$\{Q_1 \wedge Q_2 \ldots \wedge Q_n\}$$

Si los asertos de los procesos no se invalidan entre sí, entonces su composición concurrente transforma la conjunción de sus precondiciones en la conjunción de sus postcondiciones:

$$\{x == 0\}$$
$$\text{COBEGIN}$$
$$< x = x + 1; > \; \| \; < x = x + 2 >$$
$$\text{COEND}$$
$$\{x == 3\}$$



Regla de inferencia de la composición concurrente de sentencias -II

Traza de la demostración libre de interferencias:

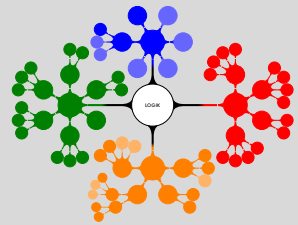
$$\{x == 0\}$$
$$\text{COBEGIN}$$
$$\{x == 0 \vee x == 2\} \qquad \{x == 0 \vee x == 1\}$$
$$< x = x + 1; > \;\|\; < x = x + 2 >$$
$$\{x == 1 \vee x == 3\} \qquad \{x == 2 \vee x == 3\}$$
$$\text{COEND}$$
$$\{x == 3\}$$

Aplicando la *regla de la no interferencia* a la expresión de asertos concurrentes anterior, la precondición de la sentencia `cobegin` ha de ser equivalente a la conjunción de las precondiciones de sus sentencias componentes

$$(x == 0 \vee x == 2) \wedge (x == 0 \vee x == 1) \equiv (x == 0)$$

Así mismo ocurre con la poscondición de `coend`

$$(x == 1 \vee x == 3) \wedge (x == 2 \vee x == 3) \equiv (x == 3)$$



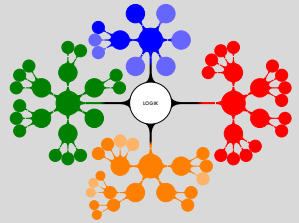
Invariante global

Invariante global: se trata de un predicado que referencia variables globales del programa que se demuestra cierto en el estado inicial de cada proceso y se mantiene cierto ante cualquier *asignación* dentro de los procesos
Ejemplo de invariante global:

- En una solución correcta del Productor-Consumidor, un invariante global sería:

$$consumidos \leq producidos \leq consumidos + 1$$

- Condición de invariante global (IG): Dado un aserto I definido a partir de las *variables compartidas* entre los procesos de un programa concurrente, puede ser considerado un IG válido si y sólo si:
  - Es cierto para los valores iniciales de las variables
  - Se mantiene cierto después de la ejecución de cada instrucción atómica *a* del programa: $\{I \wedge pre(a)\}\ a\ \{I\}$



Bibliografía

Para más información, ejercicios, bibliografía adicional, se puede consultar:

1.1. Conceptos básicos y Motivación, Palma (2003): capítulo 1

1.2. Modelo abstracto y Consideraciones sobre el hardware, Ben-Ari (2006), capítulo 2. Andrews (2000), Palma (2003): capítulo 1

1.3. Exclusión mutua y sincronización, Palma (2003), capítulo 1.

1.4. Propiedades de los Sistemas Concurrentes, Palma (2003), Capel (2022): capítulo 1

1.5. Verificación de Programas concurrentes, Andrews (2000), capítulo 2. Capel (2022): capítulo 1