Fundamentos de Lenguajes de Programación – IIC2560
Examen

7 julio 2026

1. a) Escribe un procedimiento o función que tenga 4 efectos diferentes dependiendo de si los argumentos son pasados *por valor*, *por referencia*, *por valor-resultado*, o *por nombre*.

Respuesta:

Consideremos el siguiente código, usando un arreglo y una variable de índice:

```
int x = 0;
int A[3] = {0, 0, 0};

void P(int a, int b) {
    a = 1;
    x = 2;
    b = a + 1;
}

// Llamada inicial:
// P(x, A[x]);
```

Analicemos el estado final de `x` y del arreglo `A` tras ejecutar la llamada `P(x, A[x])` según cada mecanismo de paso de parámetros:

1. **Por valor:** Se envían copias de los valores (`a = 0`, `b = A[0] = 0`). Dentro de la función, las modificaciones a `a` y `b` son estrictamente locales. Solo la asignación directa a la variable global `x=2` tiene efecto externo.
   Resultado: `x = 2`, `A = {0, 0, 0}`.

2. **Por referencia:** Se pasan punteros implícitos; `a` es un alias de `x` y `b` es un alias de `A[0]`.

   - `a = 1` modifica la variable global haciendo `x = 1`.
   - `x = 2` sobrescribe `x` haciéndolo 2 (y por ende `a` pasa a ser 2 indirectamente).
   - `b = a + 1` calcula `2 + 1 = 3` y lo asigna a `A[0]`.

   Resultado: `x = 2`, `A = {3, 0, 0}`.

3. **Por valor-resultado:** Al inicio se evalúan las direcciones (la dirección de retorno para `b` se asocia a `A[0]`) y se copian los valores locales: `a = 0`, `b = 0`.

   - Se ejecuta el cuerpo: localmente `a = 1`, globalmente `x = 2`, y localmente `b = 1 + 1 = 2`.
   - Al final se copian de vuelta en orden inverso: la variable de destino para `a` recibe el 1 (`x = 1`), y la variable de destino para `b` (`A[0]`) recibe el 2 (`A[0] = 2`).

Resultado: `x = 1, A = {2, 0, 0}`.

4. **Por nombre:** Se realiza sustitución textual implícita (Regla de Copia de Algol). Cada vez que se usa el parámetro, se re-evalúa la expresión.
   - `a = 1` $\Longrightarrow$ se ejecuta `x = 1`.
   - `x = 2` $\Longrightarrow$ `x` se actualiza a 2.
   - `b = a + 1` $\Longrightarrow$ se ejecuta `A[x] = x + 1`. Como en este punto `x=2`, se evalúa como `A[2] = 2 + 1 = 3`.

   Resultado: `x = 2, A = {0, 0, 3}`.

Se han obtenido 4 estados finales claramente diferentes demostrando los distintos mecanismos.

---

b) Considera el siguiente programa (erróneo) en C. La variable local `i` en la subrutina `foo` nunca es inicializada. Sin embargo, en algunos sistemas el programa tiene un comportamiento repetible, imprimiendo 0 1 2 3 4 5 6 7 8 9. Sugiere una explicación técnicamente válida.	`void foo() {`
	`  int i;`
	`  printf("%d ", i++);`
	`}`
	`int main() {`
	`  int j;`
	`  for (j = 1; j <= 10; j++)`
	`  foo();`
	`}`

Respuesta:

En lenguajes de programación como C, las variables como `i` se almacenan en el **Stack** dentro de un Registro de Activación. Cuando una función termina, la memoria utilizada en el stack se marca como libre, pero **no se limpia** por razones de eficiencia.

Cuando la función `main` llama a `foo` por primera vez, se reserva espacio para el registro de activación de `foo`. Como `i` no está inicializada, adopta el valor "basura" que haya quedado de antes, que casualmente en un inicio de programa suele ser 0. La función imprime 0 y luego evalúa el `i++`, por lo que el valor en esa dirección de memoria exacta se incrementa a 1.

Cuando `foo` retorna, esa memoria sigue marcada como libre. En la siguiente iteración del `for`, se invoca `foo` desde exactamente el mismo punto y con el mismo contexto temporal, por lo que **se le asigna el mismo bloque de memoria en el stack**. `i` queda mapeada a la misma dirección física que antes, donde quedó almacenado el valor 1. Imprime 1, se incrementa a 2, y el ciclo se repite. Esto hace que recicle el valor que ella misma dejó atrás y produzca el comportamiento secuencial 0 1 2 ... 9 de manera repetible durante su ejecución.

Puntaje:

- **(a) (3 ptos)** Tu código presenta 4 resultados distintos.

- **(a) (2 ptos)** Tu código presenta 3 resultados distintos.
- **(a) (1 ptos)** Tu código presenta 2 resultados distintos.
- **(a) (0 ptos)** Tu código presenta un único resultado.

- **(b) (1 pto)** Identifica que el stack no se limpia.
- **(b) (1 pto)** Concepto del valor/datos "basura" en la primera llamada
- **(b) (1 pto)** Identifica el mapeao/bloque del mismo bloque de memoria, sin marcar como usado.

2. a) En algunos lenguajes, una clase derivada puede esconder a miembros de la clase base. P.ej., en C++ podemos declarar una clase base como `public`, `protected`, o `private`:

`class B : public A { ...`

- los miembros `public` de A son miembros `public` de B
- los miembros `protected` de A son miembros `protected` de B

`class C : protected A { ...`

- los miembros `public` y `protected` de A son miembros `protected` de C

`class D : private A { ...`

- los miembros `public` y `protected` de A son miembros `private` de D

Los miembros `private` de A son *inaccesibles* para los métodos de B, C o D. Considera el impacto de clases base que sean `protected` y de clases base que sean `private` en el binding dinámico de métodos. ¿Bajo cuáles circunstancias una referencia a un instancia de la clase B, C o D puede ser asignada a una variable de tipo `A*`, y por qué?

Respuesta:

La situación depende de la clase de la que estemos hablando:

- **Para la clase B (Herencia pública):** La relación "B es un A" es públicamente visible para cualquier código en el programa. Por lo tanto, una instancia de B puede ser asignada a un `A*` **bajo cualquier circunstancia** y desde cualquier función o bloque de código.
- **Para la clase C (Herencia protegida):** Los miembros heredados de A pasan a ser `protected` en C. En consecuencia, una instancia de C solo puede ser asignada a un `A*` **exclusivamente dentro de los métodos de la clase C y dentro de los métodos de las clases derivadas de C** (además de funciones que sean *friends* de C). Desde fuera de esta jerarquía, intentar usar un C como si fuera un A arrojará un error de compilación.
- **Para la clase D (Herencia privada):** Los miembros de A pasan a ser `private` en D. Por tanto, una instancia de D puede ser asignada a un `A*` **únicamente dentro de los propios métodos de la clase D** (o sus constructores/funciones *friend*). Ni siquiera sus clases derivadas podrán ver la herencia.

El **binding dinámico** no se rompe internamente por estas modificaciones de acceso: si un upcast a `A*` es legal en el contexto actual según las reglas mencionadas arriba, el despacho dinámico se respetará normalmente, invocando correctamente la versión sobreescrita del método de la subclase.

---

b) Considera la siguiente declaración de la clase `COMPLEX` en Eiffel, en que hay dos constructores: `new_cartesian` y `new_polar`, dependiendo de si quieres crear el número complejo a partir de sus coordenadas cartesianas o polares.
Discute los compromisos (*tradeoffs*) de tiempo y espacio entre mantener los cuatro valores (x, y, ρ, θ) en el estado del objeto, o mantener sólo dos y calcular los otros *on demand* (cuando sea necesario). El operador `!!` es el equivalente en Eiffel a `new`; ejemplo de uso:

```
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(1, pi/2)
```

```
class COMPLEX
creation
    new_cartesian, new_polar
feature {ANY}
    x, y: REAL
    new_cartesian(u,v: REAL) is
    do
        x := u; y := v
    end
    new_polar(rho,theta: REAL) is
    do
        x := rho*cos(theta)
        y := rho*sin(theta)
    end
    --otros metodos public
feature {NONE}
    --metodos privados
end --class COMPLEX
```

Respuesta:

Enfoque Eager (Materialización): Mantener los 4 valores en memoria (x, y, ρ, θ)

- **Espacio:** Requiere mayor cantidad de memoria al duplicar atributos almacenados. Si se instancian colecciones masivas, el consumo de RAM aumenta linealmente con el número de objetos.
- **Tiempo y Consistencia:** El acceso de lectura es de tiempo constante $O(1)$. Sin embargo, el costo se desplaza a la escritura: cada modificación de (x, y) obliga a recalcular y actualizar sincrónicamente ρ y θ para evitar estados inconsistentes.

Enfoque Lazy (Cómputo bajo demanda): Mantener solo valores base (x, y) y calcular derivados

- **Espacio:** Diseño compacto y eficiente en memoria, almacenando solo la información mínima necesaria.
- **Tiempo:** Las operaciones de escritura son extremadamente rápidas al no requerir cálculos adicionales. El costo se traslada a la lectura: si el sistema consulta frecuentemente los valores derivados (ρ, θ), el procesador deberá ejecutar funciones matemáticas costosas (raíces, trigonométricas) en cada acceso, incrementando la carga de CPU.

La elección depende de los patrones de acceso: el enfoque *eager* es superior para aplicaciones con alta frecuencia de lectura de todos los atributos, mientras que el enfoque *lazy* es óptimo en entornos con restricciones de memoria estrictas o donde los valores derivados se consultan raramente

---

Puntaje:

- **(a) (1 pto)** Por clase B
- **(a) (1 pto)** Por clase C
- **(a) (1 pto)** Por clase D

- **(b) (0.75 ptos)** Análisis de tiempo en almacenar los 4 valores
- **(b) (0.75 ptos)** Análisis de espacio en almacenar los 4 valores
- **(b) (0.75 ptos)** Análisis de tiempo en almacenar 2 y calcular 2
- **(b) (0.75 ptos)** Análisis de tiempo en almacenar 2 y calcular 2

3. a) Explica claramente los problemas asociados al método conocido como *busy waiting*.

Respuesta:

El *busy waiting* ocurre cuando un proceso evalúa iterativamente y sin pausas si una condición se ha cumplido (como esperar un lock en la condición de un while). La iterada evaluación conlleva varios problemas para el procesador, como:

1. **Desperdicio de CPU:** En lugar de que el sistema operativo ponga a dormir el hilo bloqueado y aproveche el procesador para ejecutar otras tareas productivas, el bucle consume el 100 % de un núcleo ejecutando ciclos estériles comprobando una variable.
2. **Inversión de Prioridades (Priority Inversion):** Si un hilo de baja prioridad adquiere un recurso crítico y es interrumpido por un hilo de alta prioridad (que hace busy waiting pidiendo ese mismo recurso), el hilo de alta prioridad nunca se dormirá para soltar la CPU. Como resultado, el hilo de baja prioridad jamás logra avanzar para soltar el recurso, llevando a un virtual *deadlock* del programa.
3. **Competencia por Hardware (Bus contention):** Al leer una y otra vez la misma variable compartida alojada en la memoria, el bucle satura el bus de interconexión y de cachés del sistema, degradando drásticamente el rendimiento no solo del proceso, sino de otros núcleos y procesos que traten de acceder a la memoria física simultáneamente.

---

b) Considera una tabla de hash compartida entre varios procesos concurrentes, en que las operaciones sobre la tabla deben ser atómicas. Se puede usar un único *lock* de exclusión mutua para proteger toda la tabla, o se puede usar un esquema en que hay un *lock* por cada casilla de la tabla. ¿Cuál enfoque es probable que funcione mejor, bajo cuáles circunstancias, y por qué?

Respuesta:

Cada enfoque se adapta mejor a distintos escenarios dependiendo del nivel de paralelismo (concurrencia) exigido y del overhead de bloqueo:

Lock global:

- **Circunstancias óptimas:** Funciona mejor en aplicaciones con escasa concurrencia (pocos hilos escribiendo al mismo tiempo) o donde la tabla de hash y las inserciones sean tan diminutas/veloces que el tiempo de adquirir y soltar un lock no justifica tener miles de ellos.
- **Por qué:** Tiene la ventaja de ser muy simple de implementar y casi inmune a *deadlocks* (sin riesgo de bloqueo mutuo). Su sobrecarga de memoria es ínfima, pero con altos niveles de paralelismo creará un cuello de botella terrible: ningún proceso podrá acceder concurrentemente incluso si quieren modificar "casillas" distintas.

Lock por casilla:

- **Circunstancias óptimas:** Funciona mucho mejor en aplicaciones de alta demanda (High Performance / concurrencia masiva) en donde los procesos constantemente escriben e interrogan datos aleatoriamente distribuidos, y en tablas de hash de gran tamaño (minimizando así las colisiones entre hilos en el mismo bucket).
- **Por qué:** Permite *concurrencia real*; múltiples hilos pueden leer y modificar partes distintas de la tabla simultáneamente sin interrumpirse. Como contraparte, consume mucha más memoria alojando un semáforo (lock) por cada casilla, y puede producir deadlocks catastróficos si una operación requiere bloquear múltiples casillas a la vez (como la acción de redimensionar o hacer *rehash* del arreglo completo).

Puntaje:

- **(a) (1.5 pto)** Por problema bien explicado. (Hasta 2).
- **(b) (1.5 ptos)** Por el análises/enfoque de lock global
- **(b) (1.5 ptos)** Por el análises/enfoque de lock por casilla.

4. La sentencia

```
repeat S; until (B);
```

ejecuta `S` repetidamente hasta que `B` sea verdadero al final de alguna iteración.

a) Desarrolla una regla de inferencia para `repeat`.

Respuesta:

Debemos considerar la semántica: el comando `S` siempre se ejecuta **al menos una vez**. Después de ejecutar `S`, se evalúa la guarda `B`. Si `B` es falso ($\neg B$), el bloque vuelve a ejecutarse. Si `B` es verdadero, el ciclo finaliza.

Podemos definir la regla basándonos en un Invariante de Ciclo (Inv). Asumiendo una precondición P, la primera pasada de S debe lograr establecer el Invariante Inv. Para las subsecuentes pasadas del bucle, dado que el ciclo solo entra si se evaluó que `B` es falso, el Invariante junto con $\neg B$ deben garantizar que S restaura (preserva) el Invariante.

Formalmente, la regla de inferencia se escribe como:

$$\frac{\{P\}\ S\ \{Inv\} \quad , \quad \{Inv \wedge \neg B\}\ S\ \{Inv\}}{\{P\}\ \texttt{repeat}\ S\ \texttt{until}\ (B)\ \{Inv \wedge B\}}$$

Donde la postcondición final asegura lógicamente que el ciclo ha terminado (`B` es verdadero) y el estado válido se mantuvo (Inv).

---

b) A partir de a), desarrolla uns demostración de que `repeat` es equivalente a

```
S; while(not B) S;
```

Respuesta:

Partimos de las mismas dos premisas usadas en a):

$$\{P\}\ S\ \{Inv\} \qquad \{Inv \wedge \neg B\}\ S\ \{Inv\}$$

y mostramos que ambas construcciones son **interderivables** a partir de ellas, es decir, que las mismas premisas permiten derivar la misma tripla de Hoare para `repeat S until (B)` y para `S; while(not B) S`.

Dirección 1: derivar la tripla para `S; while(not B) S`.

Paso 1 (regla de `while`). Tomamos la segunda premisa, $\{Inv \wedge \neg B\}\ S\ \{Inv\}$, como cuerpo del ciclo `while(not B) do S`. La regla de `while` establece:

$$\frac{\{Inv \wedge \neg B\}\ S\ \{Inv\}}{\{Inv\}\ \texttt{while (not B) do}\ S\ \{Inv \wedge \neg(\neg B)\}}$$

Como $\neg(\neg B) \equiv B$, obtenemos:

$$\{Inv\}\ \texttt{while (not B) do}\ S\ \{Inv \wedge B\}$$

Paso 2 (regla de secuencia). Combinamos la primera premisa $\{P\}\ S\ \{Inv\}$ con el resultado del Paso 1:

$$\frac{\{P\}\ S\ \{Inv\} \qquad \{Inv\}\ \texttt{while (not B) do}\ S\ \{Inv \wedge B\}}{\{P\}\ S\texttt{; while (not B) do}\ S\ \{Inv \wedge B\}}$$

Con esto obtenemos:

$$\{P\}\ S\texttt{; while (not B) do}\ S\ \{Inv \wedge B\}$$

Dirección 2: derivar la tripla para `repeat S until (B)`.

Esta dirección es inmediata: la regla de `repeat` obtenida en a) es una regla primitiva que se aplica directamente sobre las mismas dos premisas:

$$\frac{\{P\}\ S\ \{Inv\} \quad , \quad \{Inv \wedge \neg B\}\ S\ \{Inv\}}{\{P\}\ \texttt{repeat}\ S\ \texttt{until}\ (B)\ \{Inv \wedge B\}}$$

obteniendo directamente:

$$\{P\}\ \texttt{repeat}\ S\ \texttt{until}\ (B)\ \{Inv \wedge B\}$$

Luego, ambas direcciones parten exactamente del mismo conjunto de premisas $\{P\}\ S\ \{Inv\}$ e $\{Inv \wedge \neg B\}\ S\ \{Inv\}$, y llegan a la misma postcondición $\{Inv \wedge B\}$:

$$\{P\}\ S\ \{Inv\},\ \{Inv \wedge \neg B\}\ S\ \{Inv\}\ \vdash\ \{P\}\ \texttt{repeat}\ S\ \texttt{until}(B)\ \{Inv \wedge B\}$$

$$\{P\}\ S\ \{Inv\},\ \{Inv \wedge \neg B\}\ S\ \{Inv\}\ \vdash\ \{P\}\ S\texttt{; while (not B)}\ S\ \{Inv \wedge B\}$$

Por lo tanto, todo triple demostrable para `repeat S until (B)` es demostrable para `S; while(not B) S`, y viceversa: las dos reglas son **interderivables** a partir del mismo conjunto de premisas.

Puntaje:

- **(a) (1 pto)** Por reconocer las iteraciones de S y momento en que se valida B.
- **(a) (1 pto)** Por las premisas.
- **(a) (1 pto)** Por las conclusión.
- **(b) (2 ptos)** Por la derivación para `S; while(not B) S`
- **(b) (1 pto)** Por la derivación para `repeat S until (B)`

5. Usando inferencia polimórfica de tipos, ¿qué tipo es asignado a cada una de las siguientes funciones?

a) `fun G(f,x){return f(f(x));}`

b) `fun H(t,x,y) {if (t(x)) return x; else return y;}`

c) `fun K(x,y) {return x;}`

Hint: Recuerda que el tipo inferido para `fun Comp(f,g,x){return f(g(x));}` es

$$(\langle S\rangle \to \langle T\rangle) \times (\langle R\rangle \to \langle S\rangle) \times \langle R\rangle \to \langle T\rangle$$

Respuesta:

Usaremos variables de tipo genéricas de la forma $\langle R\rangle, \langle S\rangle, \langle T\rangle$ para inferir paso a paso.

a) `fun G(f, x) { return f(f(x)); }`

- Asignamos un tipo genérico a `x`: $\langle R\rangle$.
- La función `f` recibe como parámetro el argumento `x`, luego el dominio de `f` debe ser $\langle R\rangle$. Supongamos que retorna un tipo $\langle S\rangle$. Entonces `f`: $\langle R\rangle \to \langle S\rangle$.
- La llamada externa es otra invocación de `f` pasándole el resultado interno. Es decir, `f` requiere recibir el tipo $\langle S\rangle$ en su dominio de entrada.
- Por lo tanto, el dominio y codominio de `f` colapsan bajo la misma restricción: $\langle R\rangle = \langle S\rangle$.
- Resultando que `f` es de tipo: $\langle R\rangle \to \langle R\rangle$.
- La función principal retorna finalmente un valor de tipo $\langle R\rangle$.

Tipo de G: $(\langle R\rangle \to \langle R\rangle) \times \langle R\rangle \to \langle R\rangle$

b) `fun H(t, x, y) { if (t(x)) return x; else return y; }`

- Asignamos un tipo genérico a `x`: $\langle R\rangle$.
- La función `t` recibe el parámetro `x`. Es la guarda de la sentencia `if`, de manera estricta debe retornar un valor tipo booleano. Por lo tanto, `t`: $\langle R\rangle \to$ bool.
- Las dos ramas condicionales del `if` deben devolver idénticamente el mismo tipo de dato. La rama `if` retorna `x` (tipo $\langle R\rangle$), lo que restringe a la rama `else` (`y`) a ser obligatoriamente del mismo tipo $\langle R\rangle$.
- El retorno global de la función `H` por ende es $\langle R\rangle$.

Tipo de H: $(\langle R\rangle \to \text{bool}) \times \langle R\rangle \times \langle R\rangle \to \langle R\rangle$

c) `fun K(x, y) { return x; }`

- Asignamos tipos completamente independientes a los argumentos porque no interactúan en operaciones. Sea `x`: $\langle R \rangle$ y `y`: $\langle S \rangle$.
- La función retorna `x`, arrojando un resultado del tipo estricto de `x` ($\langle R \rangle$).

Tipo de K: $\langle R \rangle \times \langle S \rangle \to \langle R \rangle$

Puntaje:

- **(2 ptos)** Por el tipo de a)
- **(2 ptos)** Por el tipo de b)
- **(2 ptos)** Por el tipo de c)

Debes responder la pregunta 5 más tres preguntas entre 1, 2, 3 y 4.