

# Resolución práctica 4

A continuación presentamos soluciones a ejercicios seleccionados de la [Práctica 4](#).

Como siempre, en el proceso de resolución de un problema, primero debemos dedicar un tiempo al análisis del mismo para asegurarnos de entender correctamente qué se pide. Luego, pasamos a la resolución propiamente dicha.

Recomendamos usar este documento como un elemento de ayuda y referencia. Es importante que no lean la solución sin antes haber leído y entendido el problema y sin haber dedicado un tiempo a pensar cómo resolverlo.

Por otro lado, no hay una única forma correcta de escribir una solución.

---

## 1 Estructuras

En esta práctica comenzaremos a trabajar con estructuras de datos. Veremos que ellas nos permitirán abstraer la información de forma adecuada. Además, su uso nos facilitará la realización de programas interactivos más complejos. Como siempre, será necesario que tengamos en cuenta todo lo aprendido en los ejercicios anteriores en cuanto a buenas prácticas de programación.

---

### 1.1 Ejercicio 5

En este ejercicio tenemos que hacer una versión modificada del ejercicio 4 de la práctica 3. Particularmente, utilizaremos una estructura posn como estado (en vez de un número), permitiremos que el objeto se mueva tanto vertical como horizontalmente mediante el teclado, y por último modificaremos marginalmente el comportamiento del mouse ya que ahora el objeto se dibujará en la posición donde se hizo click.

Comenzamos con el diseño de datos y algunas constantes sobre el objeto a dibujar, el fondo de la imagen y el estado inicial:

```
; El estado del sistema es una estructura posn cuyos valores
; x e y representan las coordenadas horizontal y vertical,
; respectivamente, del objeto.

; Radio del objeto.
(define OBJETO_RADIO 30)
; Color del objeto.
(define OBJETO_COLOR "red")
; Imagen del objeto. Este es el valor que vamos a usar cuando dibujemos.
(define OBJETO_IMAGEN (circle OBJETO_RADIO "solid" OBJETO_COLOR))

; Ancho del fondo.
(define FONDO_ANCHO 500)
; Alto del fondo.
(define FONDO_ALTO 500)
; Imagen del fondo. Este es el valor que vamos a usar cuando dibujemos.
(define FONDO_IMAGEN (empty-scene FONDO_ANCHO FONDO_ALTO))

; Incremento de posición al mover con teclado.
(define DELTA 10)

; Estado inicial del sistema.
```

```
(define ESTADO_INICIAL (make-posn (/ FONDO_ANCHO 2) (/ FONDO_ALTO 2)))
```

Uno de los requerimientos del ejercicio es que el objeto no se tiene que dibujar fuera de la escena. En esta resolución interpretamos ese requerimiento como que el objeto se tiene que dibujar íntegramente dentro de la escena. Para esto elegimos proceder de la siguiente manera: cada vez que transformemos el estado de nuestro sistema (las coordenadas del objeto) en uno inválido (es decir, un estado en el que el objeto se dibuje fuera de la escena) vamos a transformarlo nuevamente de modo que la coordenada resultante sea la válida más cercana.

El enunciado del ejercicio 4 de la práctica 3 nos pide que el objeto no desaparezca de la escena. Podríamos haberlo interpretado explícitamente de esa forma, permitiendo que el objeto se dibuje parcialmente dentro de la misma. ¿De qué forma habría cambiado nuestra implementación esta nueva interpretación del requerimiento? ¿Se te ocurre alguna otra interpretación?

¿Se te ocurre alguna otra forma de implementar este requerimiento?

Definimos un par de funciones auxiliares que ajustan las coordenadas horizontales y verticales del objeto y luego las combinamos en una función que nos ajuste el estado. Adicionalmente, definimos unas constantes que dependen del fondo para poder trabajar fácilmente con los límites de dibujo:

```
; Posición válida mínima sobre el eje horizontal.
(define LIMITE_IZQUIERDO OBJETO_RADIO)
; Posición válida máxima sobre el eje horizontal.
(define LIMITE_DERECHO (- FONDO_ANCHO OBJETO_RADIO))
; Posición válida mínima sobre el eje vertical.
(define LIMITE_SUPERIOR OBJETO_RADIO)
; Posición válida máxima sobre el eje vertical.
(define LIMITE_INFERIOR (- FONDO_ALTO OBJETO_RADIO))

; acomodar_horizontal: Number -> Number
; Dada una coordenada horizontal, la acomoda al intervalo
; válido [LIMITE_IZQUIERDO, LIMITE_DERECHO].
(check-expect (acomodar_horizontal 300) 300)
(check-expect (acomodar_horizontal 15) 30)
(check-expect (acomodar_horizontal 471) 470)
(define (acomodar_horizontal x)
  (cond
    [(< x LIMITE_IZQUIERDO) LIMITE_IZQUIERDO]
    [(< LIMITE_DERECHO x) LIMITE_DERECHO]
    [else x]))

; acomodar_vertical: Number -> Number
; Dada una coordenada vertical, la acomoda al intervalo
; válido [LIMITE_SUPERIOR, LIMITE_INFERIOR].
(check-expect (acomodar_vertical 100) 100)
(check-expect (acomodar_vertical 20) 30)
(check-expect (acomodar_vertical 2020) 470)
(define (acomodar_vertical y)
  (cond
    [(< y LIMITE_SUPERIOR) LIMITE_SUPERIOR]
    [(< LIMITE_INFERIOR y) LIMITE_INFERIOR]
    [else y]))

; acomodar_estado: Estado -> Estado
; Dado un estado, acomoda cada una de las coordenadas al
; intervalo válido correspondiente.
(check-expect (acomodar_estado (make-posn 20 10)) (make-posn 30 30))
```

```
(check-expect (acomodar_estado (make-posn 150 300)) (make-posn 150 300))
(check-expect (acomodar_estado (make-posn 300 1000)) (make-posn 300 470))
(define (acomodar_estado estado)
  (make-posn (acomodar_horizontal (posn-x estado)) (acomodar_vertical (posn-y estado))))
```

Las funciones `acomodar_vertical` y `acomodar_horizontal` comparten mucho código. ¿Qué podríamos haber hecho al respecto?

A continuación, definimos el manejador de pantalla que simplemente extrae las coordenadas de nuestro estado para dibujar el objeto sobre el fondo:

```
; manejador_pantalla: Estado -> Image
; Dibuja la pantalla
(define (manejador_pantalla estado)
  (place-image OBJETO_IMAGEN (posn-x estado) (posn-y estado) FONDO_IMAGEN))
```

Ahora definimos el manejador de teclado. Para esto, transformamos el estado según la tecla presionada devolviendo una nueva estructura `posn` y a ese valor le aplicamos la función `acomodar_estado` para obtener un estado válido:

```
; Incremento de posición al mover con teclado.
(define DELTA 10)

; manejador_teclado: Estado String -> Estado
; Transforma el estado apropiadamente según la tecla presionada
; y lo acomoda a un estado válido:
; * "right" incrementa la coordenada x en DELTA unidades.
; * "left" decrementa la coordenada x en DELTA unidades.
; * "down" incrementa la coordenada y en DELTA unidades.
; * "up" decrementa la coordenada y en DELTA unidades.
; * cualquier otra tecla mantiene el estado.
(check-expect (manejador_teclado (make-posn 470 470) "right") (make-posn 470 470))
(check-expect (manejador_teclado (make-posn 40 333) "left") (make-posn 30 333))
(check-expect (manejador_teclado (make-posn -1000 -1000) "down") (make-posn 30 30))
(check-expect (manejador_teclado (make-posn 200 150) "up") (make-posn 200 140))
(check-expect (manejador_teclado (make-posn 65536 -32767) "a") (make-posn 470 30))
(check-expect (manejador_teclado (make-posn 120 120) " ") (make-posn 250 250))
(define (manejador_teclado estado tecla)
  (acomodar_estado
   (cond
    [(key=? tecla "right") (make-posn (+ (posn-x estado) DELTA) (posn-y estado))]
    [(key=? tecla "left") (make-posn (- (posn-x estado) DELTA) (posn-y estado))]
    [(key=? tecla "down") (make-posn (posn-x estado) (+ (posn-y estado) DELTA))]
    [(key=? tecla "up") (make-posn (posn-x estado) (- (posn-y estado) DELTA))]
    [(key=? tecla " ") ESTADO_INICIAL]
    [else estado])))
```

El último manejador que nos queda es el del mouse. Procedemos de forma similar a la elegida con el manejador de teclado: transformamos el estado de acuerdo al evento del mouse y a ese nuevo valor le aplicamos la función `acomodar_estado` para obtener un estado válido:

```
; manejador_mouse: Estado Number Number String -> Estado
; Transforma el estado apropiadamente según el evento de
; mouse y lo acomoda a un estado válido. Si se hace click
; ("button-down") en algún lugar de la pantalla entonces
; transforma el estado y posteriormente lo acomoda.
(check-expect (manejador_mouse (make-posn -100 -1000) 67 231 "button-down") (make-posn 67 231))
(check-expect (manejador_mouse (make-posn 420 210) 10 700 "button-down") (make-posn 30 470))
(check-expect (manejador_mouse (make-posn 100 100) 343 451 "move") (make-posn 100 100))
(define (manejador_mouse estado x y boton)
```

```
(acomodar_estado (if (mouse=? boton "button-down") (make-posn x y) estado)))
```

Finalmente, combinamos todo en la expresión big-bang:


```
(big-bang
  ESTADO_INICIAL
  [to-draw manejador_pantalla]
  [on-key manejador_teclado]
  [on-mouse manejador_mouse])
```

---

## 1.2 Ejercicio 6

En este ejercicio se nos pide que movamos una mosca aleatoriamente, con la condición de que esta pueda ser atrapada a través del cursor del mouse. El enunciado ya nos dice que el estado del sistema será un posn, representando la posición actual de la mosca. También nos indica cuál será el estado inicial.

Comenzamos definiendo algunas constantes necesarias, y haciendo el diseño de datos.

```
; Imagen a dibujar


(define MOSCA )

; Ancho y alto de la escena
(define ANCHO 700)
(define ALTO 700)
; Fondo de la escena
(define FONDO (empty-scene ANCHO ALTO))

; Diseño de datos.
; Representamos posiciones mediante una estructura posn
; El estado del sistema es un objeto de tipo posn
; que representa la posición de la mosca en la escena

; Estado inicial
(define INICIAL (make-posn (/ ANCHO 2) (/ ALTO 2)))
```

Por cada tick del reloj la mosca se deberá mover aleatoriamente una distancia DELTA, tanto vertical como horizontalmente. Es decir, para cada dirección (horizontal o vertical), la mosca podrá desplazarse DELTA o -DELTA unidades, dando lugar a cuatro resultados posibles para el nuevo estado. Para realizar esto, diseñamos algunas funciones.

```
; elegir-random : Number Number -> Number
; Dados dos números, devuelve uno de ellos en forma aleatoria
; Ejemplos (no se puede usar check-expect de manera directa)
; Entrada {5,6}, posible salida 5
; Entrada {3,8}, posible salida 8
; Entrada {7,7}, salida 7
(define (elegir-random a b) (if (= (random 2) 1) a b))

; Cantidad de píxeles que puede moverse
; la mosca tanto horizontal como verticalmente.
(define DELTA 10)

; mover : Number -> Number
```

```

; Dado un número, le suma o le resta DELTA unidades,
; decidiendo aleatoriamente.
(define (mover n) (+ n (elegir-random DELTA (- DELTA))))

; reloj : Estado -> Estado
; Dada una posición, devuelve el estado siguiente,
; permitiendo a la mosca decidir, para cada dirección,
; si se moverá en un sentido o en el otro.
; Los movimientos realizados siempre son de DELTA unidades.
(define (reloj p) (make-posn (mover (posn-x p)) (mover (posn-y p))))

; interpretarSimple : Estado -> Imagen
; Dado un estado, devuelve una imagen
; con la mosca en la posición determinada
; por el mismo.
(define (interpretarSimple p) (place-image MOSCA (posn-x p) (posn-y p) FONDO))

; Expresión big-bang
(big-bang INICIAL
  [to-draw interpretarSimple]
  [on-tick reloj])

```

Ahora debemos modificar la función anterior para asegurarnos de que la mosca no salga fuera de escena. Para hacerlo, definimos algunas constantes y funciones auxiliares.

```

; Constantes para asegurar que la mosca entra en la escena
; Límites horizontales
(define LIZQ (/ (image-width MOSCA) 2))
(define LDER (- ANCHO (/ (image-width MOSCA) 2)))
; Límites verticales
(define LSUP (/ (image-height MOSCA) 2))
(define LINF (- ALTO (/ (image-height MOSCA) 2)))

; acomodar : posn -> posn
; Dada una posición, si esta implica que la mosca
; se dibujará fuera de la escena, devuelve una nueva
; donde la mosca aparece completa.
(check-expect (acomodar INICIAL) INICIAL)
(check-expect (acomodar (make-posn 0 0)) (make-posn LIZQ LSUP))
(check-expect (acomodar (make-posn ANCHO ALTO)) (make-posn LDER LINF))
(define (acomodar p)
  (make-posn (cond [(< (posn-x p) LIZQ) LIZQ]
                  [(> (posn-x p) LDER) LDER]
                  [else (posn-x p)])
    (cond [(< (posn-y p) LSUP) LSUP]
          [(> (posn-y p) LINF) LINF]
          [else (posn-y p)])))

```

Piense en dónde debería llamar a la función acomodar para obtener el efecto deseado.

Para saber si la mosca es atrapada, en cada click del mouse se deberá comparar la posición del evento con la de la mosca. Si se encuentran a una distancia menor a un GAMMA, entonces la mosca se considerará atrapada, y se pasará a un estado final, distinto al resto. Elegiremos la posición `(-1, -1)`, pues nunca se alcanza dicho estado en el funcionamiento del programa.

Notemos que la función que determina si la mosca fue atrapada debe ser la que está asociada a la cláusula `on-mouse`, pues es la única que toma como argumentos la posición del cursor. A su vez, también sabemos que, dicha función, lo único que puede hacer es *transformar* el estado. Por lo tanto, necesitamos codificar en el estado si la mosca fue atrapada o no. Esto se puede lograr a través de una posición que no es alcanzable de otro modo. Luego, si la mosca está en la posición `(-1, -1)`, entonces es porque fue atrapada.

A continuación, definimos constantes y diseñamos funciones para tal fin.

```
; Distancia que determina si se atrapa la mosca
(define GAMMA (/ (image-width MOSCA) 2))

; Estado final
(define FINAL (make-posn -1 -1))

; distancia: posn posn -> Number
; Dadas dos posiciones, devuelve la distancia
; entre los puntos que representan
(check-expect (distancia (make-posn 6 9) (make-posn 9 13)) 5)
(check-expect (distancia (make-posn 0 0) (make-posn 6 0)) 6)
(check-expect (distancia (make-posn 6 9) (make-posn 6 0)) 9)
(define (distancia p q)
  (sqrt (+ (sqr (- (posn-x p) (posn-x q)))
           (sqr (- (posn-y p) (posn-y q))))))

; mouse : Estado Number Number String -> Estado
; Dados el estado actual, y la posición y tipo de evento
; del mouse, devuelve el nuevo estado de acuerdo a las
; siguientes condiciones:
; Si el evento es "button-down" y se produjo a distancia
; menor que GAMMA, devuelve estado FINAL
; En otro caso, no modifica el estado
(check-expect (mouse INICIAL (posn-x INICIAL) (posn-y INICIAL) "button-down") FINAL)
(check-expect (mouse INICIAL ANCHO ALTO "button-down") INICIAL)
(check-expect (mouse INICIAL ANCHO ALTO "move") INICIAL)
(define (mouse p x y s)
  (cond [(string=? s "button-down") (if (< (distancia p (make-posn x y)) GAMMA) FINAL p)]
        [else p]))
```

Ahora debemos modificar la función asociada a la cláusula to-draw, de manera que al alcanzar el estado final, imprima el texto "MOSCA ATRAPADA" en pantalla.

```
; texto para el estado final
(define MSJ-FINAL (text "MOSCA ATRAPADA" 50 "blue"))

; final? : Estado -> Boolean
; dado un estado, devuelve #true si es el estado final
(check-expect (final? INICIAL) #false)
(check-expect (final? FINAL) #true)
(define (final? p)
  (and (= (posn-x FINAL) (posn-x p))
       (= (posn-y FINAL) (posn-y p))))

; interpretar : Estado -> Imagen
; Dado un estado, devuelve una imagen
; con la mosca en la posición determinada
; por el mismo. Si es el estado final,
; imprime "MOSCA ATRAPADA" en la pantalla
(define (interpretar p)
  (cond
    [(final? p) (place-image MSJ-FINAL (/ ANCHO 2) (/ ALTO 2) FONDO)]
    [else (place-image MOSCA (posn-x p) (posn-y p) FONDO)]))
```

Por último, se debe configurar la cláusula stop-when, para que al alcanzar el estado final, termine el programa.

```
; Expresión big-bang
```

```
(big-bang INICIAL
  [to-draw pantalla]
  [on-tick reloj]
  [on-mouse mouse]
  [stop-when final? pantalla])
```

De esta forma, el ejercicio queda resuelto acorde a su enunciado. Sin embargo, reflexionemos sobre la expresión `big-bang` anterior.

A la cláusula `to-draw` le asociamos la función `pantalla`. A la cláusula `stop-when` le asociamos un predicado `final?` y, nuevamente, la función `pantalla`. Sin embargo, podríamos asociarle a cada cláusula una función distinta.

La función asociada a la cláusula `stop-when`, se llamará únicamente cuando el predicado `final?` sea verdadero, es decir, cuando la mosca haya sido atrapada. Por lo tanto, en ese caso, sabemos que queremos graficar la imagen final. A su vez, la función asociada a la cláusula `to-draw`, se llamará únicamente cuando el estado no sea el final. Por lo tanto, en ese caso, sabemos que queremos graficar a la mosca en la posición que indica el estado.

Siguiendo este razonamiento, otra posible forma de resolver el ejercicio es como se muestra a continuación.

```
; interpretarSimple : Estado -> Imagen
; Dado un estado, devuelve una imagen
; con la mosca en la posición determinada
; por el mismo.
(define (interpretarSimple p) (place-image MOSCA (posn-x p) (posn-y p) FONDO))

; interpretarFinal : Estado -> Imagen
; Dado un estado, devuelve la imagen final.
(define (interpretarFinal p) (place-image MSJ-FINAL (/ ANCHO 2) (/ ALTO 2) FONDO))

; Expresión big-bang
(big-bang INICIAL
  [to-draw interpretarSimple]
  [on-tick reloj]
  [on-mouse mouse]
  [stop-when final? interpretarFinal])
```

Notemos que en este caso, las funciones que dibujan no se encargan de calcular si el estado es el final o no, ya que de eso se está ocupando la expresión `big-bang`.

---

## 1.3 Ejercicio 12

Este ejercicio nos pide calcular el Índice de Masa Corporal de una persona, para lo cual debemos crear una estructura de datos `persona`, con los datos dados y además poder utilizarla para trabajar con esta.

Como siempre, comenzaremos con el diseño.

Como bien nos aclara el ejercicio, tenemos 5 campos. Nombre y apellido, que representaremos con una string. Podrían ser dos y estaría bien, pero dado que el espacio es un carácter válido, podemos poner todo dentro de una sola y que esté espaciado. Un número que representa el peso, una string que representa la medida, un número que representa la estatura y un string que representa la medida de la estatura. En resumen, nos quedaría algo similar a esto:

```
; Diseño de datos
(define-struct persona [nombre peso unipes estatura uniest])
; una persona es (make-persona String Number String Number String)
; interpretación: representamos una persona mediante su nombre,
```

```
; peso y estatura, agregando campos para la unidad de medida de estos
; dos últimos valores.
```

```
; Representamos estaturas y pesos mediante números
; Representamos nombres mediante strings
; Representamos unidades de estatura mediante strings ("m" "cm")
; Representamos unidades de peso mediante strings ("kg" "g")
```

Con esto ya tendríamos diseñada la estructura persona y con ello todas las funciones que Racket crea para interactuar con cualquier estructura. Es decir, la función `make-persona`, como constructor, todos los selectores para trabajar con cada campo de la estructura, como por ejemplo `persona-nombre` que me permite trabajar con el nombre de una persona en dicha estructura, y el predicado `persona?` que me permite preguntar si el tipo de dato que recibo es efectivamente una estructura persona y no un número, imagen, posn o cualquier otra cosa.

Es una buena práctica, tanto para hacer testeos de nuestra función, como para saber si lo que definimos está funcionando, definir unos **casos ejemplo de nuestra nueva estructura de datos** para poder ver si nos funciona.

```
; Ejemplos de personas
(define SILVIA (make-persona "Silvia Perez" 68 "kg" 1.64 "m"))
(define FER (make-persona "Fernando Aguirre" 72 "kg" 179 "cm"))
(define NICO (make-persona "Nicolás Zanarini" 24000 "g" 1.23 "m"))
(define ANA (make-persona "Ana Gomez" 65000 "g" 170 "cm"))
```

Recordemos que `check-within` hace un check dentro de un margen, en este caso al tener un número fraccionario, resulta imposible testear de forma exacta, así que nos basta que esté en un margen de 0.1 del resultado esperado, dejando espacio a una aproximación

Luego pasaremos a diseñar y resolver la función `imc` que en caso de ser tipo persona, calcula el índice de masa corporal de una persona, en este caso, guardada en una estructura persona.

Para ello deberemos chequear que lo que recibimos es efectivamente del tipo Persona utilizando el predicado de la estructura, para luego mediante una función auxiliar calcular el índice de la forma que sea necesaria, dependiendo de las unidades.

Es conveniente **modularizar el código**, es decir, separar en funciones según distintas utilidades. No solo es más organizado sino que además nos permite aislar potenciales errores y testear en forma separada. Por esto resulta importante hacer la función `imcaux` en forma separada.

```
; imc : persona -> Number/String
; dada una persona, devuelve su índice de masa corporal;
; para cualquier otro objeto, devuelve "Tipo de dato inválido"
(check-within (imc FER) 22.47 0.1)
(check-within (imc ANA) 22.49 0.1)
(check-expect (imc "Hola") "Tipo de dato inválido")
(define (imc pers)
  (cond
    [(persona? pers) (imcaux pers)]
    [else "Tipo de dato inválido"])))

; imcaux : persona -> Number
; dada una persona, devuelve su índice de masa corporal;
(check-within (imcaux ANA) 22.49 0.1)
(check-within (imcaux SILVIA) 25.28 0.1)
(define (imcaux pers)
  (cond
    [(and (string=? (persona-unipes pers) "kg") (string=? (persona-uniest pers) "m"))
```



```

(/ (persona-peso pers) (sqr (persona-estatura pers)))
[(and (string=? (persona-unipes pers) "kg") (string=? (persona-uniest pers) "cm"))
 (/ (persona-peso pers) (sqr (* 0.01 (persona-estatura pers))))]
[(and (string=? (persona-unipes pers) "g") (string=? (persona-uniest pers) "m"))
 (/ (* 0.001 (persona-peso pers)) (sqr (persona-estatura pers)))]
[(and (string=? (persona-unipes pers) "g") (string=? (persona-uniest pers) "cm"))
 (/ (* 0.001 (persona-peso pers)) (sqr (* 0.01 (persona-estatura pers)))))]))

```

Con la idea de modularizar, una solución alternativa sería crear una función que resuelva el Índice de Masa Corporal para kilogramos y metros, y crear funciones auxiliares que transformen, en caso de ser necesario, la unidad en la que estamos trabajando.

```

; Dado un peso en kilogramos y una estatura en metros,
; calcula el Índice de Masa Corporal en kg/m^2
; imc-kg-m : Number Number -> Number
(check-within (imc-kg-m 72 1.79) 22.47 0.1)
(define (imc-kg-m peso estatura) (/ peso (sqr estatura)))

; Dado un peso, evalúa si está en kilogramos. Si no lo está,
; lo convierte de gramos a kilogramos.
; toKg : Number String -> Number
(check-expect (toKg 8 "kg") 8)
(check-expect (toKg 1200 "g") 1.2)
(define (toKg peso unidad) (if (string=? unidad "kg") peso (* 0.001 peso)))

; Dado un peso, evalúa si está en metros. Si no lo está,
; lo convierte de centímetros a metros
; toM : Number String -> Number
(check-expect (toM 1.7 "m") 1.7)
(check-expect (toM 184 "cm") 1.84)
(define (toM medida unidad) (if (string=? unidad "m") medida (* 0.01 medida)))

; Dada una persona, devuelve su índice de masa corporal
; imc2 : persona -> Number
(check-within (imc2 ANA) 22.49 0.1)
(check-within (imc2 SILVIA) 25.28 0.1)
(define (imc2 pers)
  (imc-kg-m (toKg (persona-peso pers) (persona-unipes pers))
    (toM (persona-estatura pers) (persona-uniest pers))))

```