Examen Final

Lenguajes de Programación

López Soto Ramses Antonio

20 de agosto de 2021

1. Menciona, explica y ejemplifica los siguientes conceptos:

   a. Sintaxis

   Consiste en cumplir las convenciones dadas por el lenguaje, es decir, deben de cumplirse las reglas estipuladas previamente para que programa en el lenguaje fuente esté bien escrito. Un ejemplo claro de esto sería una simple suma *+ 8 9*.

   b. Semántica

   Se encarga de darle un sentido a las expresiones escritas en el lenguaje, es decir, la semántica le da un significado al programa. Un ejemplo sencillo seria $x = 7$.

   c. Convenciones de programación (*Idioms*)

   Son aquellas reglas que no están predeterminadas en el lenguaje; son aquellas convenciones que no son obligatorias pero facilita la lectura del programa. Un ejemplo de ello, los nombres de los métodos pueden ser en formato *camelCase* (prenderTele()) o *snake_ case* (prender_tele()).

   d. Lenguaje objetivo

   Es aquel lenguaje en el cual se implementa un programa; un ejemplo claro, Python.

   e. Lenguaje anfitrión

   Es aquel lenguaje que es usado como "fuente"por el intérprete y/o compilador, por ejemplo, el compilador de Python está escrito en C, por lo que el lenguaje anfitrión de Python es C.

2. Explica los paradigmas (a) orientado a objetos, (b) funcional de los lenguajes de programación, menciona dos lenguajes que pertenezcan a cada paradigma.

   a. Paradigma Orientado a Objetos (*POO*)

   Consiste en la construcción, diseño e implementación de programas basados en objetos que poseen algún comportamiento, y además poseen características especificas como la herencia, clases, encapsulamiento, entre otros. Siendo este perteneciente a un enfoque imperativo donde se busca responder el CÓMO; este paradigma está presente en lenguajes como C++ y JAVA.

b. Paradigma Funcional (*PF*)

Se basa en gran medida por definiciones de funciones y procedimientos que tiene sus raíces en el Cálculo Lambda. Además, posee un enfoque declarativo donde se desea responder el QUÉ; este paradigma está presente en lenguajes como HASKELL y Earlang.

1.5

3. Explica todos los tipos de análisis que se requieren para generar código ejecutable dado el código fuente.

a. Análisis Léxico

Esta etapa se encarga del flujo de caracteres de un programa, es decir, descompone una cadena en átomos o lexemas que permite realizar un análisis más preciso y detallado, por lo que el lexer devuelve parejas del tipo *<clase, lexema>*, es decir, *tokens*.

b. Análisis Sintáctico

Esta fase del análisis es la encargada de verificar que las reglas establecidas por la gramática del lenguaje se cumplan para poder construir un árbol de sintaxis abstracta (*ASA*), el parser es quien lo realiza.

c. Análisis Semántico

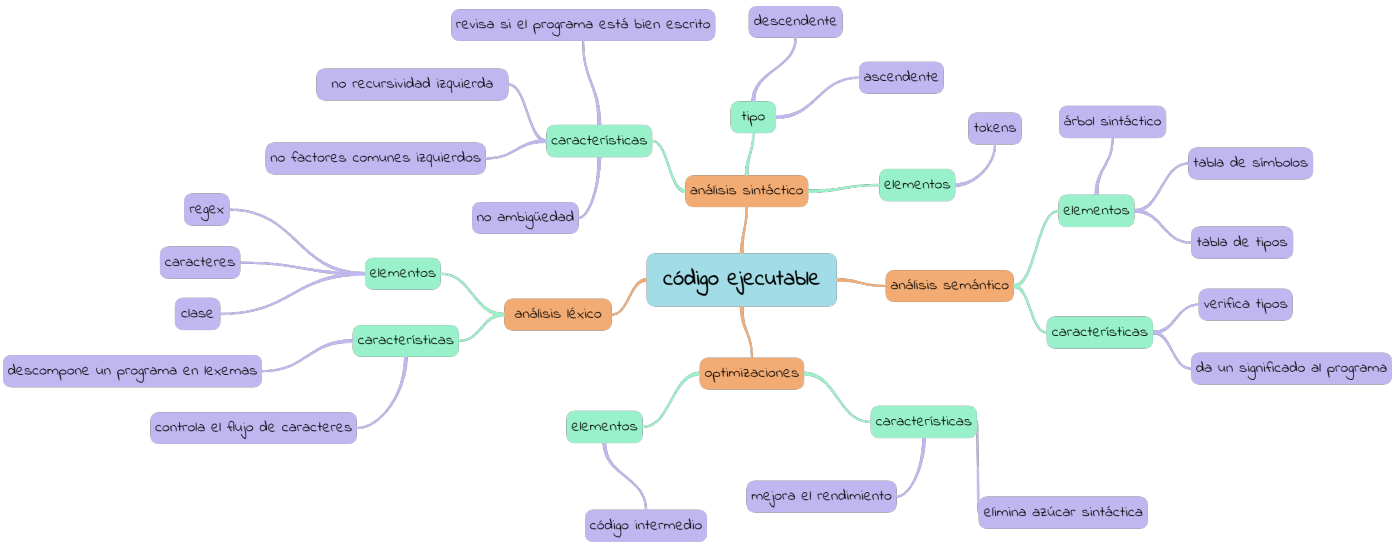
La etapa más importante, debido a que su tarea principal es la de revisar que el programa tenga un sentido y le da un significado específico. El analizador semántico es quien realiza todo este y devuelve un *ASA* con significado.

Existen más tipos.

d. Optimizaciones

Esta etapa se encarga de la transformación de código (eliminar azúcar sintáctica) para para mejorar el rendimiento del programa en tiempo de compilación.

A continuación se muestra un mapa mental con las características principales de los tipos de análisis anteriores:



4. Explica los tres tipos de identificadores que hay en un lenguaje de programación y explica por qué la evaluación de las expresiones depende únicamente de las variables libres. Da un ejemplo usando la gramática del lenguaje *FWAE* visto en clase.

   a. De ligado

   Son aquellos identificadores que se les asigna un valor específico que puede ser utilizado de alguna forma dentro de un alcance.

   b. Ligados

   Estos identificadores son aquellos que se asocian a un identificador de ligado con el mismo nombre dentro de un alcance.

   c. Libres

   Los identificadores libres son aquellos que no poseen un identificador de ligado dentro de un alcance específico.

   Con esto, es importante saber que la evaluación de una expresión es dependiente de los identificadores libres debido a que estos tienen un papel sumamente importante, ya que determinan si dicha expresión puede o no ser evaluada, es decir, si se posee algún identificador libre, el programa n pude terminar su ejecución a causa de algo que nunca se declaró. A continuación, se ejemplifica lo anterior:

   ```
   {with {x 25} {expt x {+ z y}}}
   ```

   Es claro que el ejemplo anterior, posee los tres tipos de identificadores: *{x 25}* es de ligado por lo que x es ligada, pero lo que causa conflicto es esta expresión es la subexpresión *{+ z y}* pues no está claro cuales son los valores de z y y debido a que son libres, entonces el cuerpo de la expresión no puede ser evaluado porque la ejecución se detiene lanzando un error al no saber quiénes son z y y.

5. Explica por qué el algoritmo de sustitución visto en clase es de $O(n^2)$.

   R. La complejidad del algoritmo $O(n^2)$, esto se debe a que en cualquier expresión se tienen n identificadores de ligado, esto implica que el algoritmo va a recorrer la expresión por cada identificador de ligado ya sea que realice la sustitución en el cuerpo de la expresión en caso de que encuentre un identificador ligado o no, es decir, haya sustitución o no el algoritmo va a realizar el mismo proceso para cada uno los identificadores de ligado. Esto implica que si el programa posee n variables, tendrá

   $$n + (n-1) + (n-2) + ... + 2 + 1 = \frac{n(n-1)}{2}$$

   sustituciones en total. En pocas palabras, hace el recorrido n veces para n identificadores y esto da una complejidad de $O(n^2)$.

6. Transforma el siguiente código usando direcciones léxicas y evalúalo.

```
{with {a 1}
  {with {b 2}
    {with {c {+ b 3}}
      {with {d {+ a b}}
        {with {e 4}
           {+ e {+ d {+ c {+ b a}}}}}}}}}
```

Su representación correspondiente en direcciones léxicas (*Índices de Bruijn*):

```
{with 1
  {with 2
    {with {+ <:0> 3}
      {with {+ <:2> <:1>}
        {with 4
           {+ <:0> {+ <:1> {+ <:2> {+ <:3> <:4>}}}}}}}}}
```

La evaluación se ve como sigue:

```
{with 2
  {with {+ <:0> 3}
    {with {+ 1 <:1>}
      {with 4
         {+ <:0> {+ <:1> {+ <:2> {+ <:3> 1}}}}}}}}}

  {with {+ 2 3}
    {with {+ 1 2}
      {with 4
         {+ <:0> {+ <:1> {+ <:2> {+ 2 1}}}}}}}}

  {with {+ 1 2}
    {with 4
       {+ <:0> {+ <:1> {+ {+ 2 3} {+ 2 1}}}}}}

    {with 4
       {+ <:0> {+ {+ 1 2} {+ {+ 2 3} {+ 2 1}}}}}}

       {+ 4 {+ {+ 1 2} {+ {+ 2 3} {+ 2 1}}}}

       {+ 4 {+ {+ 1 2} {+ {+ 2 3} 3}}

       {+ 4 {+ {+ 1 2} {+ 5 3}}
```

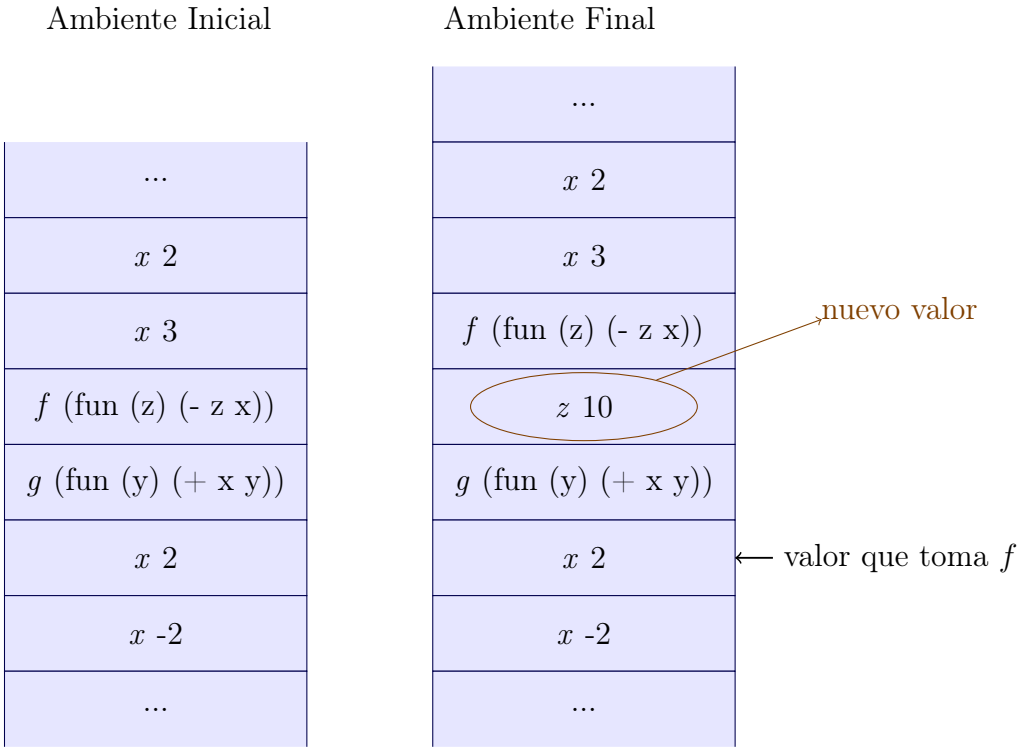
```
{+ 4 {+ {+ 1 2} 8}}

{+ 4 {+ 3 8}}

{+ 4 11}

15
```

7. Evalúa la siguiente expresión utilizando (a) alcance estático y (b) alcance dinámico, es necesario poner el ambiente (*Stack*) de ejecución usando representación de listas para poder evaluar.

```
{with {x -2}
  {with {x 2}
    {with {g { fun {y} {+ x y}}}
      {with {f {fun {z} {- z x}}}
        {with {x 3}
          {with {x 2}
            {f 10}}}}}}}
```

a. Alcance Estático

La representación por medio de ambientes es la siguiente:

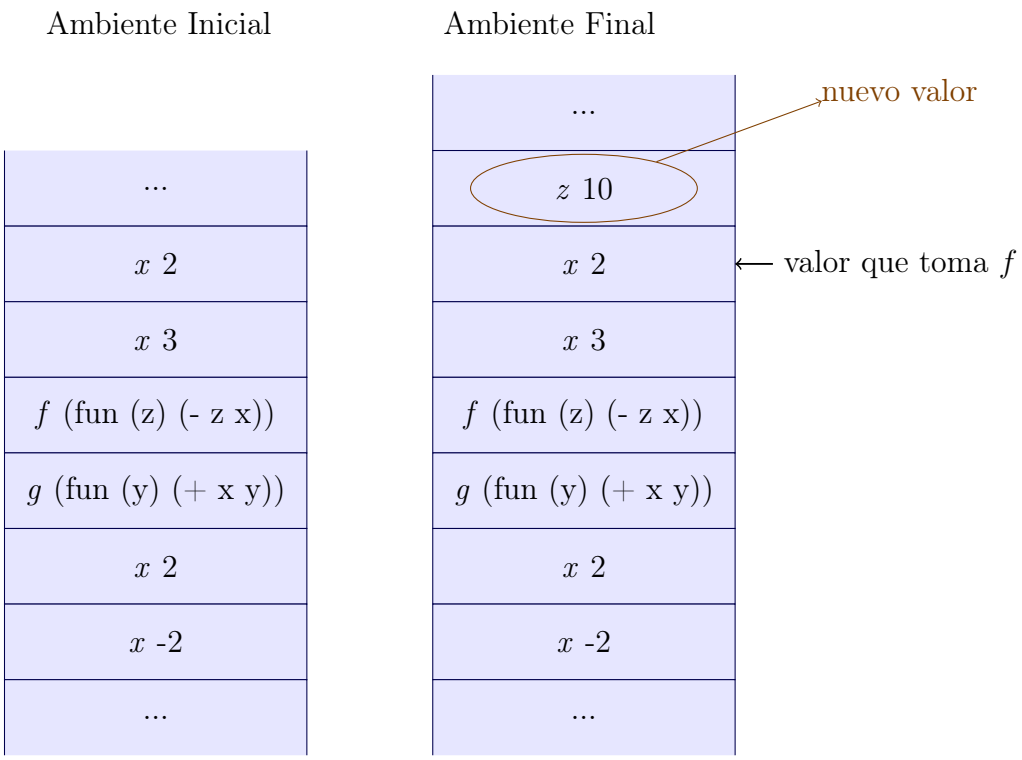


La representación anterior ejemplifica la siguiente evaluación:

```
> (f 10)
> ((fun (z) (- z x)) 10)
> (- 10 x)
> (- 10 2)
> 8
```

b. Alcance Dinámico

La representación por medio de ambientes es la siguiente:



La representación anterior ejemplifica la siguiente evaluación:

```
> (f 10)
> ((fun (z) (- z x)) 10)
> (- 10 x)
> (- 10 2)
> 8
```

8. Evalúa la siguiente expresión usando:

a. Evaluación glotona y alcance estático.

b. Evaluación glotona y alcance dinámico.

c. Evaluación perezosa y alcance estático.

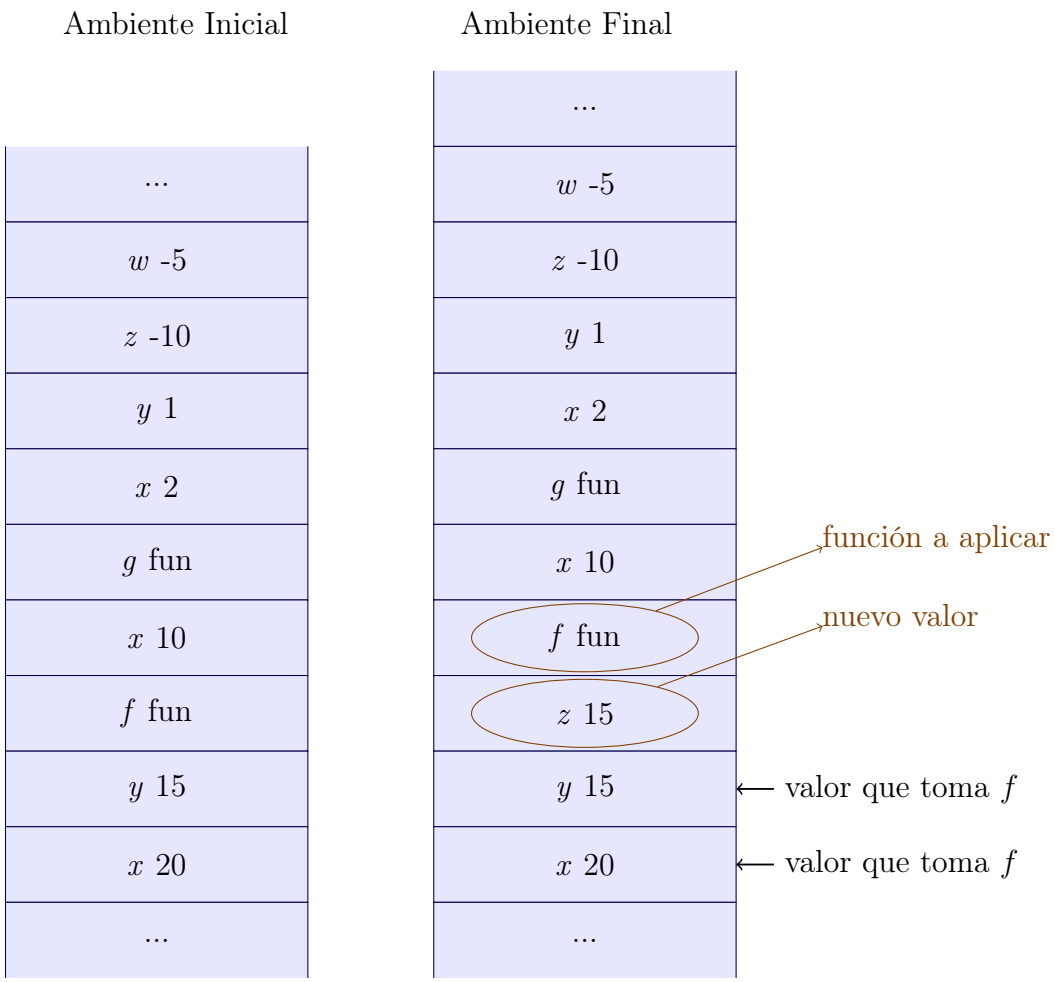
d. Evaluación perezosa y alcance dinámico.

y escribe el ambiente con representación de *Stack* cuando la evaluación es perezosa y otro cuando la evaluación es glotona.

```
{with {x {+ 10 10}}
  {with {y { - x 5}}
    {with {f { fun {z} {+ x {+ y z}}}}
      {with {x {+ 5 5}}
        {with {g { fun {w} {+ x {+ y w}}}}
          {with {x 2}
            {with {y 1}
              {with {z -10}
                {with {w -5}
                  {f 15}}}}}}}}}}
```

a. Evaluación glotona y alcance estático

La representación por medio de ambientes es la siguiente:



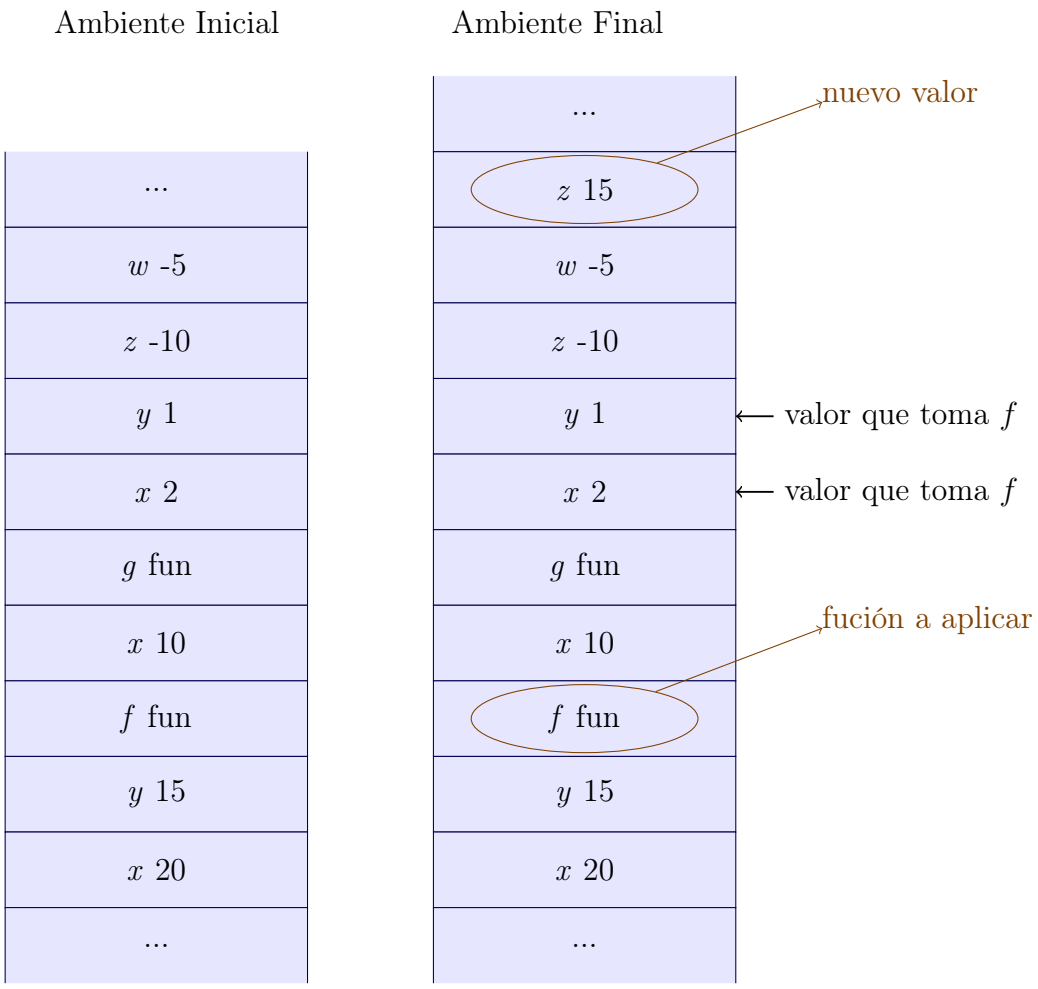
La representación anterior ejemplifica la siguiente evaluación:

```
> (f 15)
```

```
> ((fun (z) (+ x (+ y z))) 15)
> (+ x (+ y 15))
> (+ x (+ 15 15))
> (+ 20 30)
> 50
```

b. Evaluación glotona y alcance dinámico

La representación por medio de ambientes es la siguiente:

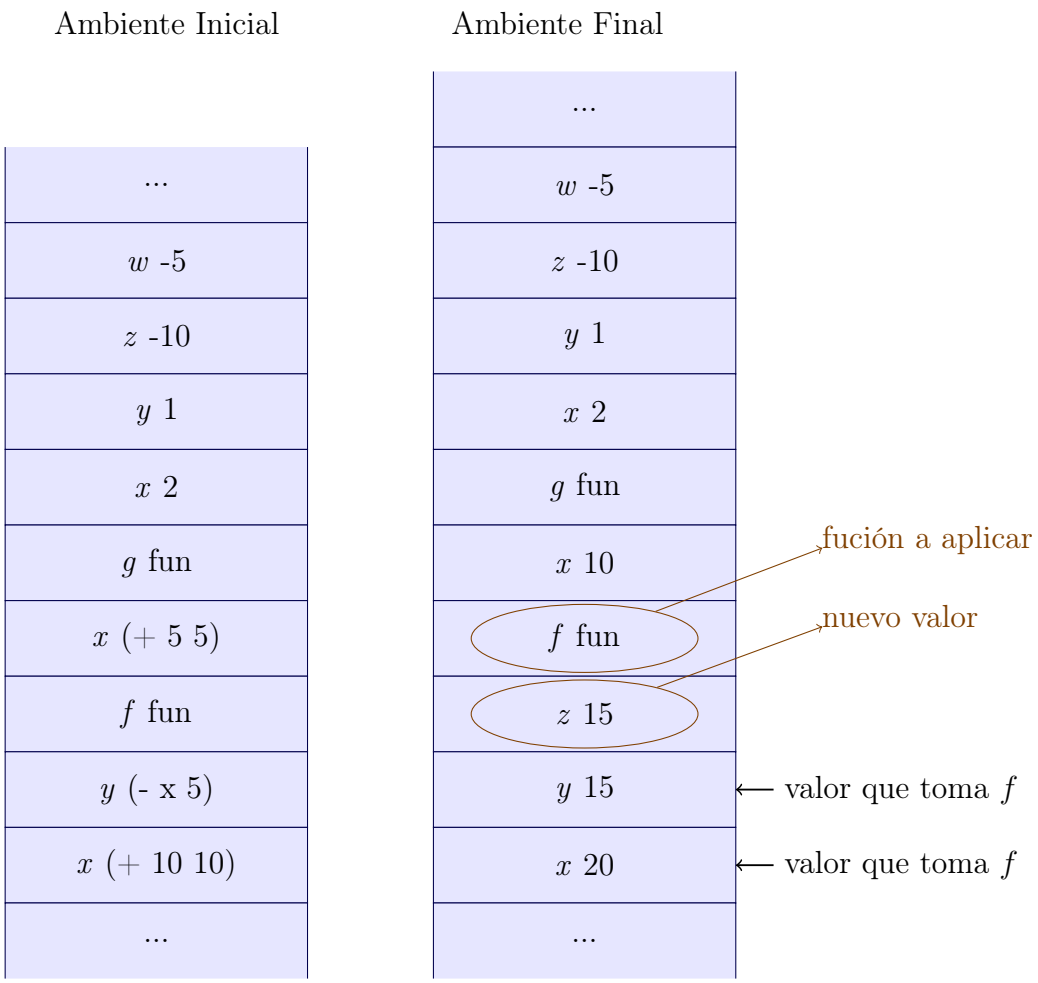


La representación anterior ejemplifica la siguiente evaluación:

```
> (f 15)
> ((fun (z) (+ x (+ y z))) 15)
> (+ x (+ y 15))
> (+ x (+ 1 15))
> (+ 2 16)
> 18
```

c. Evaluación perezosa y alcance estático

La representación por medio de ambientes es la siguiente:

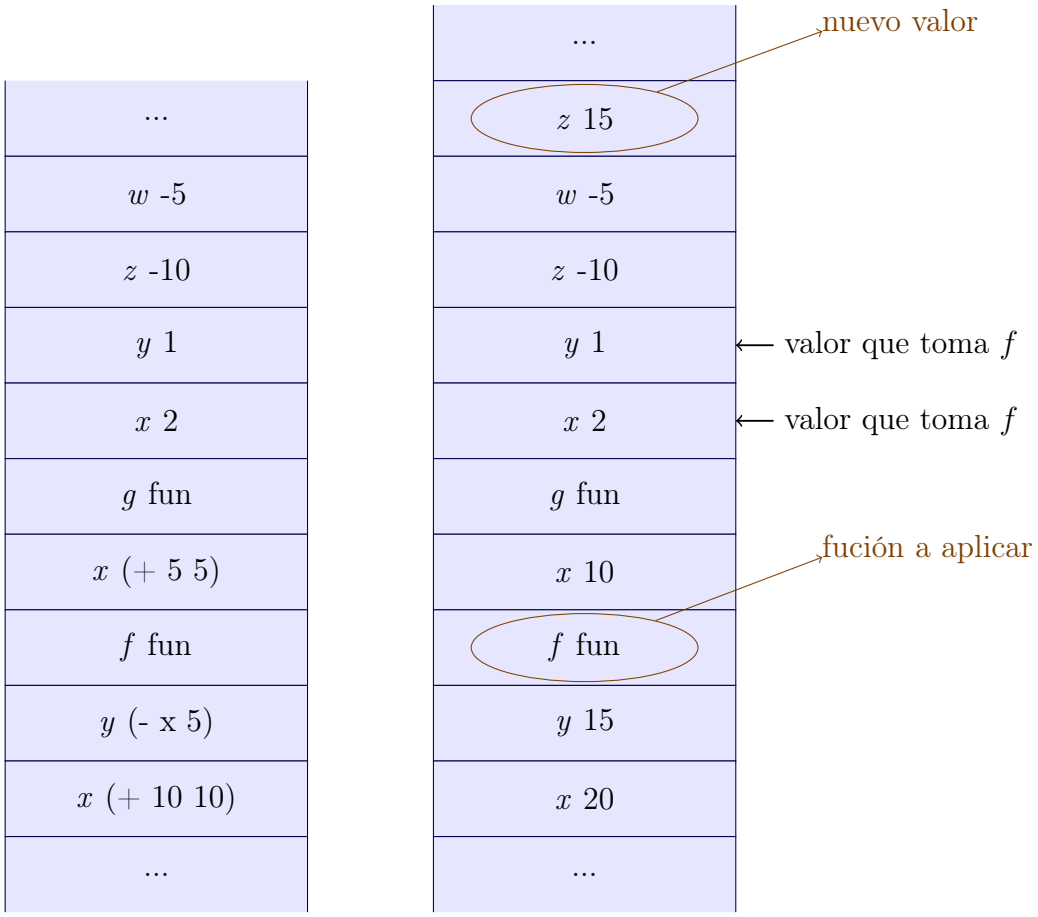


La representación anterior ejemplifica la siguiente evaluación:

```
> (f 15)
> ((fun (z) (+ x (+ y z))) 15)
> (+ x (+ y 15))
> (+ x (+ (- x 5) 15))
> (+ (+ 10 10) (+ (- x 5) 15))
> (+ 20 (+ (- x 5) 15))
> (+ 20 (+ 15 15))
> (+ 20 30)
> 50
```

d. Evaluación perezosa y alcance dinámico

La representación por medio de ambientes es la siguiente:



La representación anterior ejemplifica la siguiente evaluación:

```
> (f 15)
> ((fun (z) (+ x (+ y z))) 15)
> (+ x (+ y 15))
> (+ x (+ 1 15))
> (+ 2 (+ 1 15))
> (+ 2 16)
> 18
```

9. ¿Por qué fue necesario introducir *closures* para evaluar expresiones con alcance estático en la función *interp* vista en clase?

R. Esto es debido a que le facilita la carga del trabajo al intérprete ya que es posible almacenar una lista de parámetros formales, funciones específicas y ambientes de sustitución, y con estos la implementación del alcance estático puede ser posible y exitosa.

10. ¿De qué otra manera se podría implementar en un lenguaje alcance estático si no hubiera *closures*?

R. Probablemente, haciendo uso de listas ligadas, pero podría sufrir utal cambio que la eficiencia sería peor que con closures.

11. Dentro del Cálculo Lambda, evalúa cada una de las siguientes expresiones usando -reducciones. Si alguna tiene Forma Normal, especifícala.

a. $(\lambda x.x)(\lambda x.xxx)$

$(\lambda x.x)(\lambda x.xxx) \rightarrow_\beta (\lambda x.xxx)$

No hay más sustituciones posibles, entonces la expresión ya se encuentra en Forma Normal.

b. $(\lambda x.(\lambda y.yxw)z)u$

$(\lambda x.(\lambda y.yxw)z)u \rightarrow_\beta (\lambda y.yuw)z \rightarrow_\beta zuw$

No hay más sustituciones posibles, entonces la expresión ya se encuentra en Forma Normal.

c. $(\lambda x.\lambda y.\lambda z.x)(yz)$

$(\lambda x.\lambda y.\lambda z.x)(yz) \rightarrow_\beta (\lambda y.\lambda z.yz)$

No hay más sustituciones posibles, entonces la expresión ya se encuentra en Forma Normal.

12. ¿Qué es y para qué sirve el Combinador Y? Da un ejemplo.

R. El combinador Y se define como

$$Y =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

y, además es de punto fijo.

Este es usado usado en gran medida para definir recursión en las funciones que lo requieran. Un ejemplo claro del uso del combinador Y es el siguiente:

```
1
2  ;; Definicion del combinador Y
3  (define Y (lambda (f) ((lambda (x) (f (x x))) (lambda (x) (f (x x))))))
4
5  ;; Uso del combinador Y
6  (let ([rev (Y (lambda (rev)
7                  (lambda (l)
8                      (if (empty? l)
9                          '()
10                         (append (rev (cdr l)) (list (car l)))))))])
11    rev (list 1 2 3))
12
```

En el ejemplo anterior siempre se genera una nueva aplicación de la función **rev**, por lo que nunca acaba.

13. Explica 3 características de la evaluación perezosa y a qué se le conoce como punto estricto en lenguajes perezosos.

- Las operaciones sólo se realizan cuando es sumamente estricto.
- Consume más espacio al almacenar operaciones sin evaluar.
- Permite definir estructuras de datos infinitas.

Además, se tiene lo que se le conoce como *punto estricto* que es una propiedad que obliga a las expresiones a reducirse a un valor específico (si es que existe).

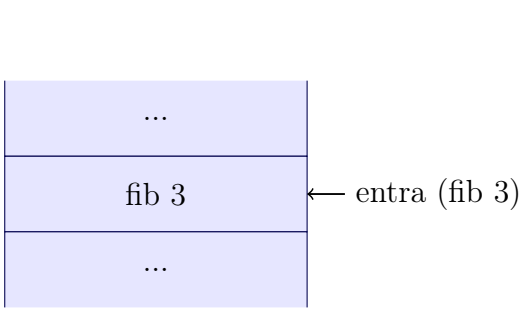
14. ¿Cuáles son las implicaciones que trae consigo la recursión de cola en comparación con la recursión normal en cuanto a la memoria utilizada en tiempo de ejecución?

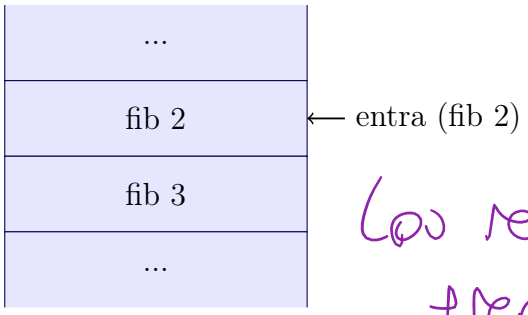
R. La recursión de cola trae consigo una serie de mejores de lo que la recursión normal puede ofrecer. Una de estas es que la recursión de cola brinda una serie de optimizaciones debido a que no consume mucha memoria puesto que solo hace uso de un registro de activación, donde la recursión normal gasta más recursos e inclusive puede haber un desbordamiento de la pila de ejecución.

15. Escribe dos funciones en *Racket* que calculen el *fibonacci* de un números usando (a) recursión y (b) recursión de cola. Pon el número total de registros de activación usados en ambos casos y escribe el *Stack* con los registros de activación en cada llamada tanto de la función recursiva como de la que implementes usando recursión de cola cuando se llame a ambas funciones con el argumento 3.

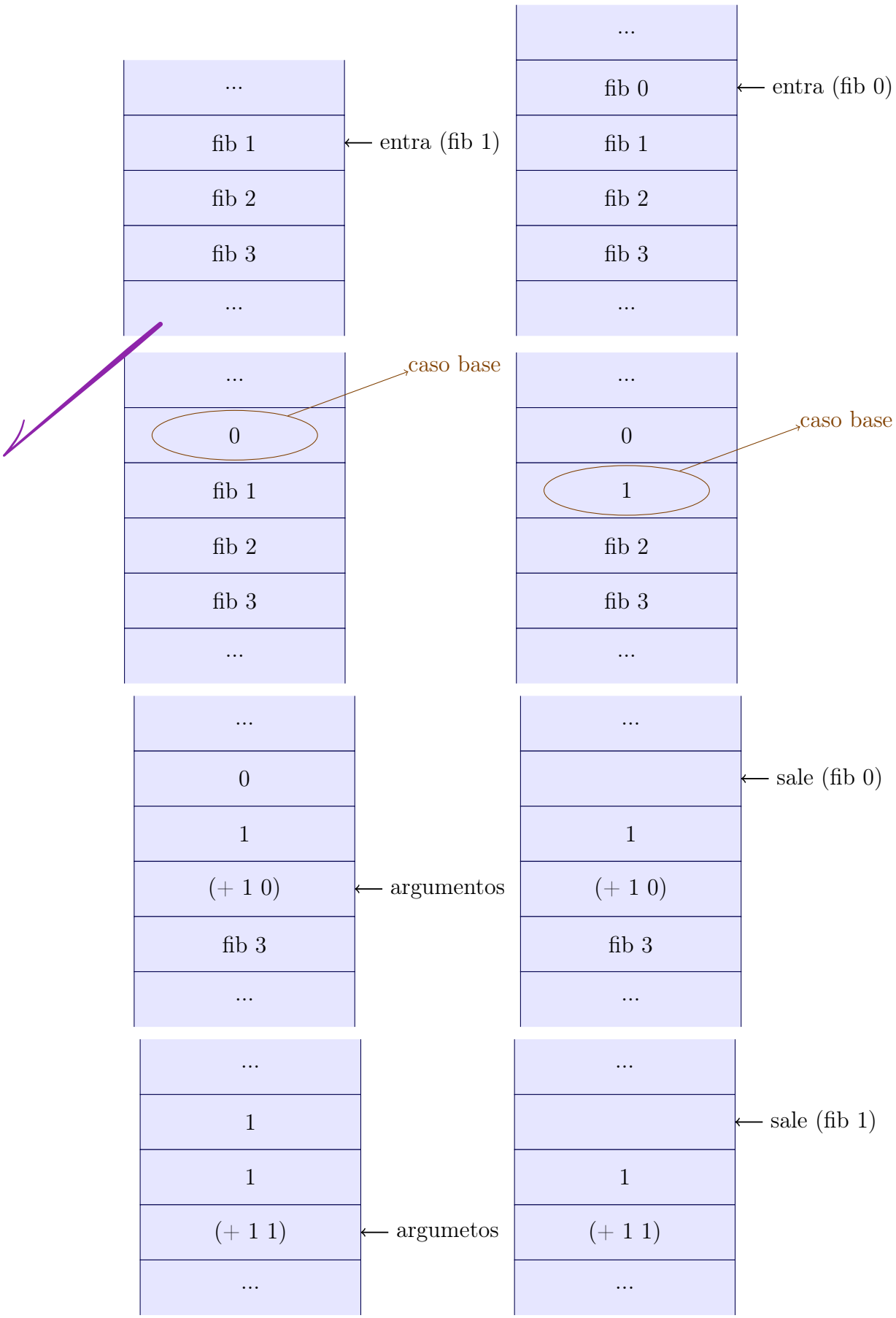
Recursión Normal

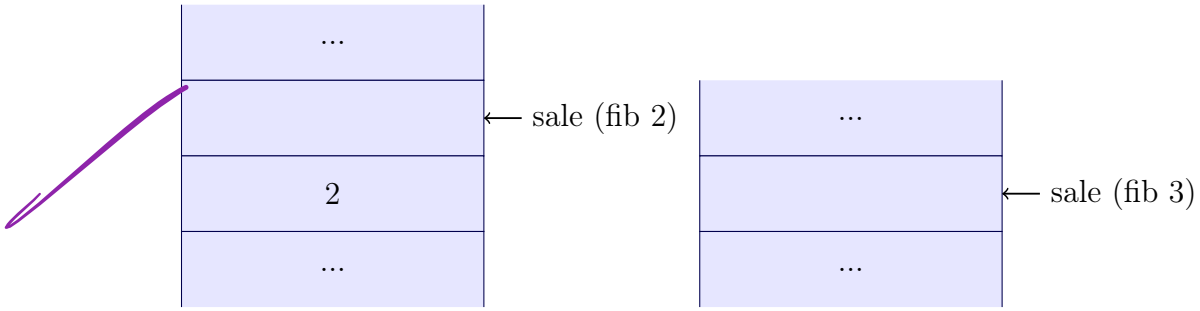
```
1
2  ;; Obtiene el fibonacci de un numero
3  ;; fibonacciR :: number --> number
4  (define (fibonacciR n)
5    (cond
6      [(equal? n 0) 0]
7      [(equal? n 1) 1]
8      [else (+ (fibonacciR (- n 1)) (fibonacciR (- n 2)))]))
9
```





Los registros tienen más información.

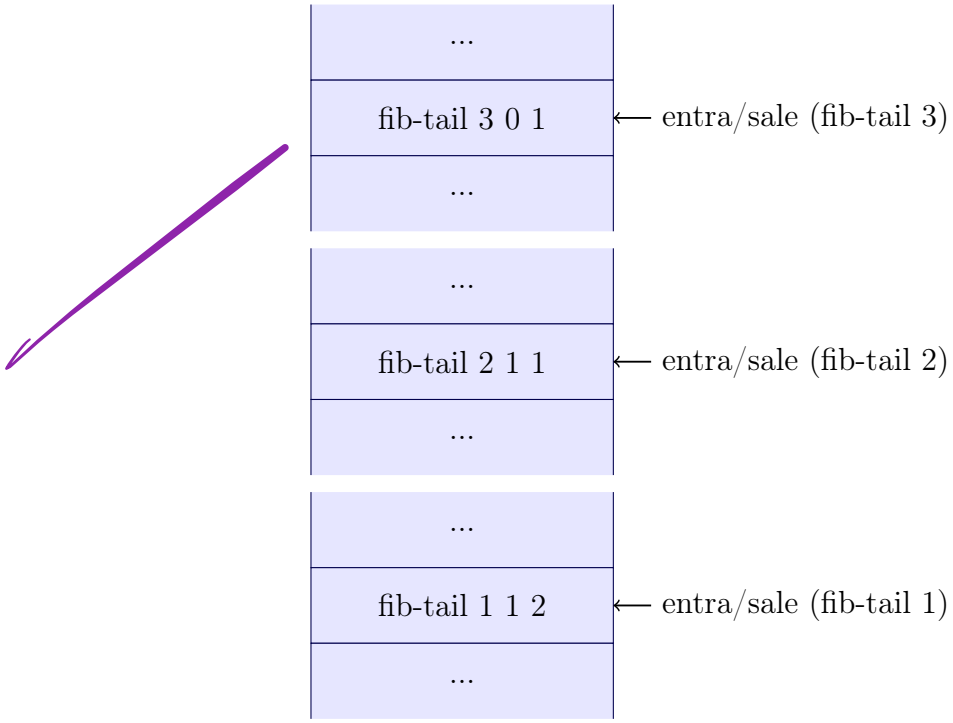

...
fib 1
fib 2
fib 3
...
entra (fib 1)
...
fib 0
fib 1
fib 2
fib 3
...
entra (fib 0)
...
0
fib 1
fib 2
fib 3
...
caso base
...
0
1
fib 2
fib 3
...
caso base
...
0
1
(+ 1 0)
fib 3
...
argumentos
...
1
(+ 1 0)
fib 3
...
sale (fib 0)
...
1
1
(+ 1 1)
...
argumetos
...
1
1
(+ 1 1)
...
sale (fib 1)



∴ `(fib 3) = 2`

Recursión de Cola

```
1
2  ;; Obtiene el fibonacci de un numero
3  ;; fibonacciT :: number --> number
4  (define (fibonacciT n)
5    (fibonacci-tail n 0 1))
6
7  ;; Obtiene el fibonacci de un numero
8  ;; fibonacci-tail :: number number number --> number
9  (define (fibonacci-tail n acc1 acc2)
10   (cond
11     [(equal? n 0) acc1]
12     [(equal? n 1) acc2]
13     [else (fibonacci-tail (sub1 n) acc2 (+ acc1 acc2))]))
14
```



∴ `(fib-tail 3 0 1) = 2`

16. Define y ejemplifica el concepto de continuación (*Continuation*) ¿qué es? ¿cuáles son sus características ¿cómo es su comportamiento en memoria? su utilidad, etc.

R. Las continuaciones consisten en un abstracción donde las pilas de ejecución de un programa se pueden almacenar como un valor y/o pasarse como un parámetro, básicamente, se *materializa*. Estas generalmente son usadas en diversas tareas, tales como corrutinas o manejo de excepciones, así como facilitar la tarea de diseñar un servidor de web. Además, de reduce espacio en memoria debido a que simplifica algunas tareas. Un ejemplo sencillo sería: `(+ 1 (let/cc k (k 3)))`

17. Evalúa los siguientes códigos y pon la función con representación $\lambda\uparrow$ que estás evaluando:

    a.
    ```
    (+ (call/cc (lambda (k)
           (k (begin
                 (set ! +8_{woo} k)
                 (display "inside body")
                 5)))
        8)
    ```

    ```
    (+ (lambda ↑ (k) (k (begin (set! +8_{woo} k) (display "inside body") 5))) 8)

    > (+ (k (begin (set! +8_{woo} k) (display "inside body") 5)) 8)
    "inside body"
    > (+ (k 5) 8)
    > (+ 5 8)
    > 13
    ```

    b.
    ```
    (+ 8 (call/cc (lambda (k)
              (+ 1 (+ 2 (+ 3 (k 2)))))))
    ```

    ```
    (+ 8 (lambda ↑ (k) (k (+ 8 2))))

    > (k (+ 8 2))
    > (k 10)
    > 10
    ```

18. Implementa la siguiente función recursiva usando *CPS*.

    ```
    (define (filter-pos l)
      (cond
        [(empty? l) empty]
        [else (if (> (car l) 0)
                  (cons (car l) (filter-pos (cdr l)))
                  (filter-pos(cdr l)))]))
    ```

Para poder transformar una función a **CPS**, es útil seguir la siguiente convención:

Función (Recursiva) → *Función (Recursion de Cola)* → *Función (CPS)*

- Versión Recursiva

```
1
2  ;; Obtiene los numeros positivos de una lista
3  ;; filter-pos :: (listof number) --> (listof number)
4  (define (filter-pos l)
5    (cond
6      [(empty? l) empty]
7      [else (if (> (car l) 0)
8                (cons (car l) (filter-pos (cdr l)))
9                (filter-pos (cdr l)))]))
10
```

- Versión Recursiva de Cola

```
1
2 ;; Obtiene los numeros positivos de una lista
3 ;; filter-posT :: (listof number) --> (listof number)
4 (define (filter-posT l)
5   (filter-pos-tail l empty))
6
7 ;; Obtiene los numeros positivos de una lista
8 ;; filter-pos-tail :: (listof number) list --> (listof number)
9 (define (filter-pos-tail l acc)
10   (cond
11     [(empty? l) acc]
12     [else (if (> (car l) 0)
13               (filter-pos-tail (cdr l) (cons (car l) acc))
14               (filter-pos-tail (cdr l) acc))]))
15
```

- Versión CPS

```
1
2  ;; Obtiene los numeros positivos de una lista
3  ;; filter-pos-cps :: (listof number) --> (listof number)
4 (define (filter-pos-cps xs)
5   (filter-pos/k l (lambda (x) x)))
6
7 ;; Obtiene los numeros positivos de una lista
8 ;; filter-pos/k :: (listof number) lambda --> (listof number)
9 (define (filter-pos/k l k)
10   (cond
11     [(empty? l) (k empty)]
12     [else (if (> (car l) 0)
13               (filter-pos/k (cdr l)
14                             (lambda (x) (k (cons (car l) x))))
15               (filter-pos/k (cdr l) k))]))
16
```

19. ¿Por qué se introdujeron las operaciones con cajas al intérprete? Explica a profundidad.

R. Porque resultan sumamente útiles ya que nos permiten recuperar y modificar la información almacenada en ellas además de que proveen una memoria para que las cajas sean mutables. En otras palabras, la cajas ayudan a simular Estado, haciendo que el lenguaje sea mucho más flexible.

20. Evalúa la siguiente expresión usando:

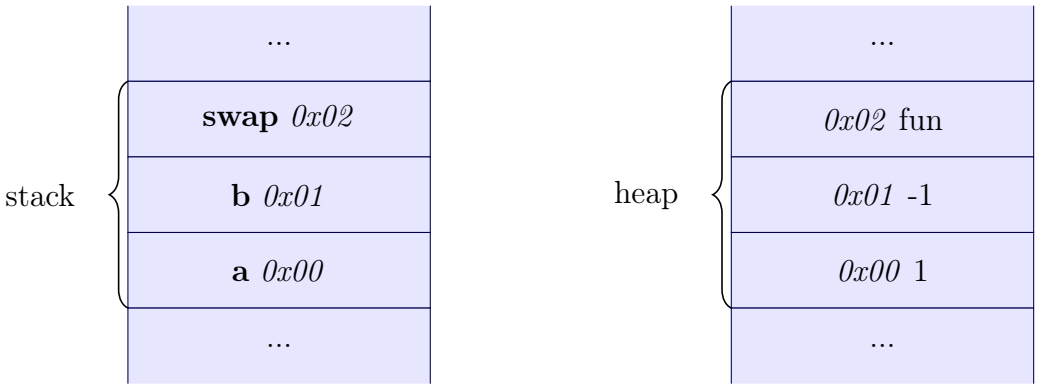
    a. Paso de parámetros por valor
    b. Paso de parámetros por referencia

```
{with {{a 1}
       {b -1}
       {swap {fun {x y}
                  {with {tmp x}
                    {seqn {set x y}
                          {set y tmp}}}}}}
    {seqn {swap a b}
          {- a b}}}
```

Por Valor

Se inicializan el *Stack* y el *Heap*:

```
{a 1}
    {b -1}
      {swap {fun {x y}
                 {with {tmp x}
                    {seqn {set x y}
                          {set y tmp}}}}}
```

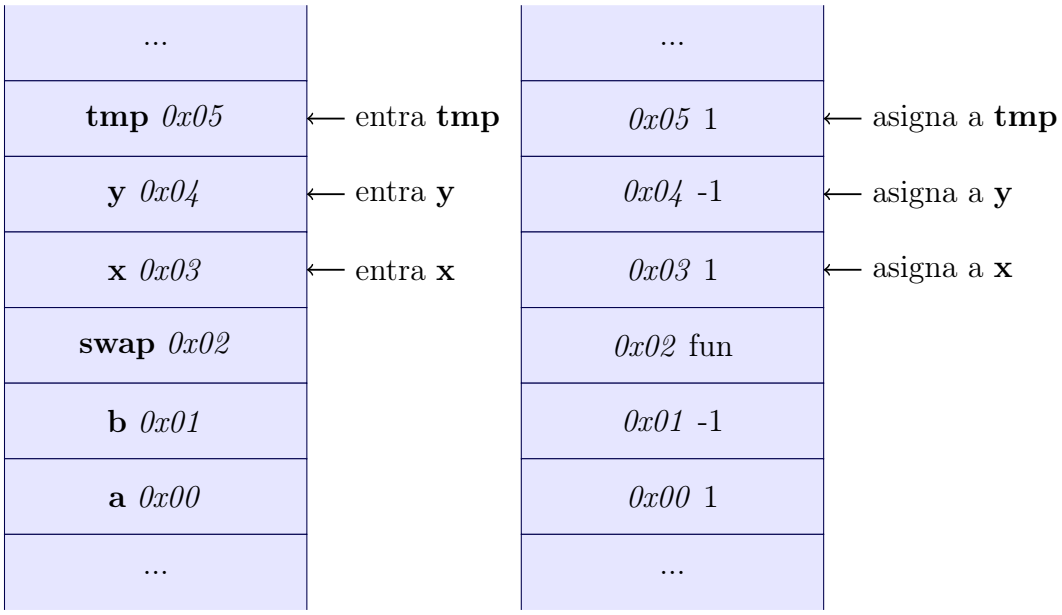


Se tiene que:

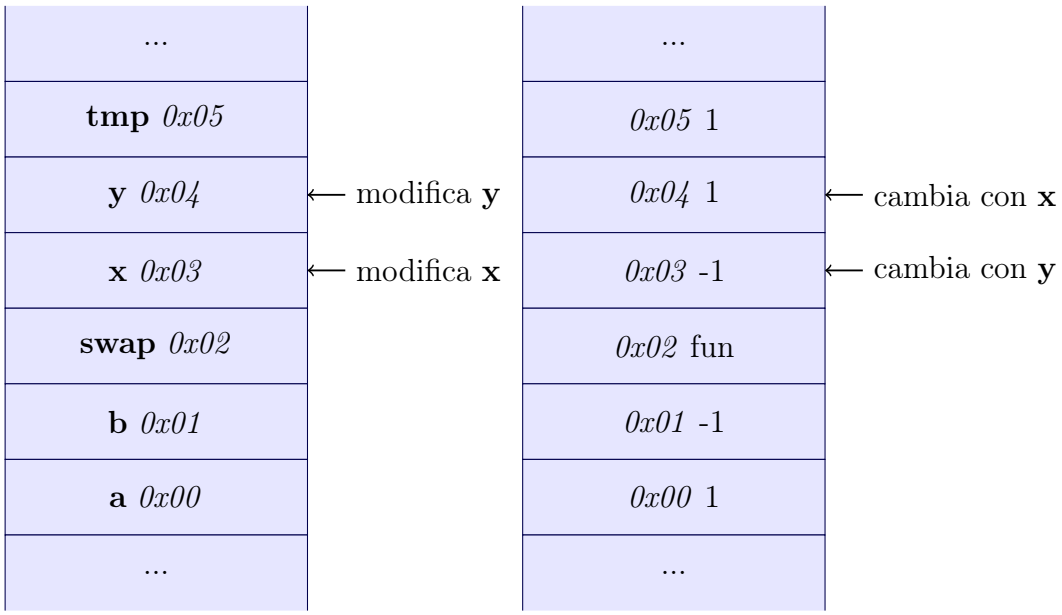
parámetros reales: a b

parámetros formales: x y

```
{swap a b}

{fun {x y}
    {with {tmp x}
            {seqn {set x y}
                {set y tmp}}}}
```




```
{seqn {swap a b}
      {- a b}}
```



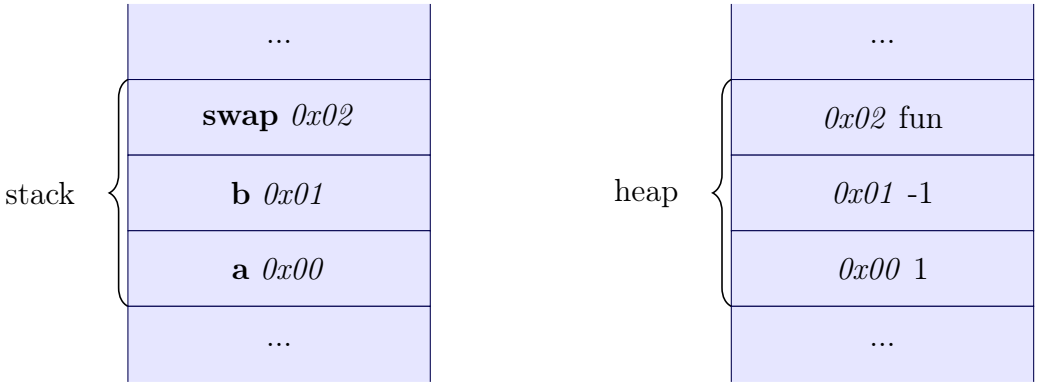
```
> (- a b)
> (- 1 -1)
> 2
```

∴ el resultado en 2.

Por Referencia

Se inicializan el *Stack* y el *Heap*:

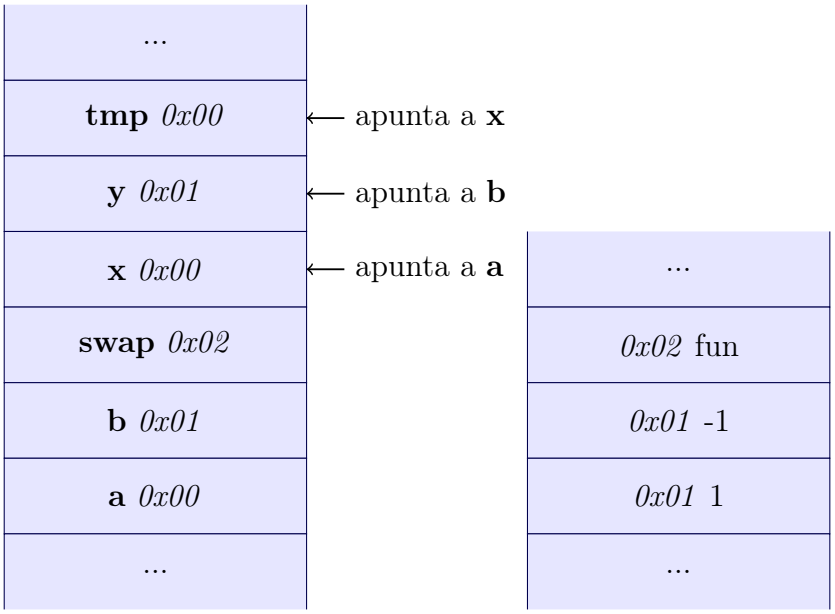
```
{a 1}
    {b -1}
      {swap {fun {x y}
                 {with {tmp x}
                    {seqn {set x y}
                          {set y tmp}}}}}
```



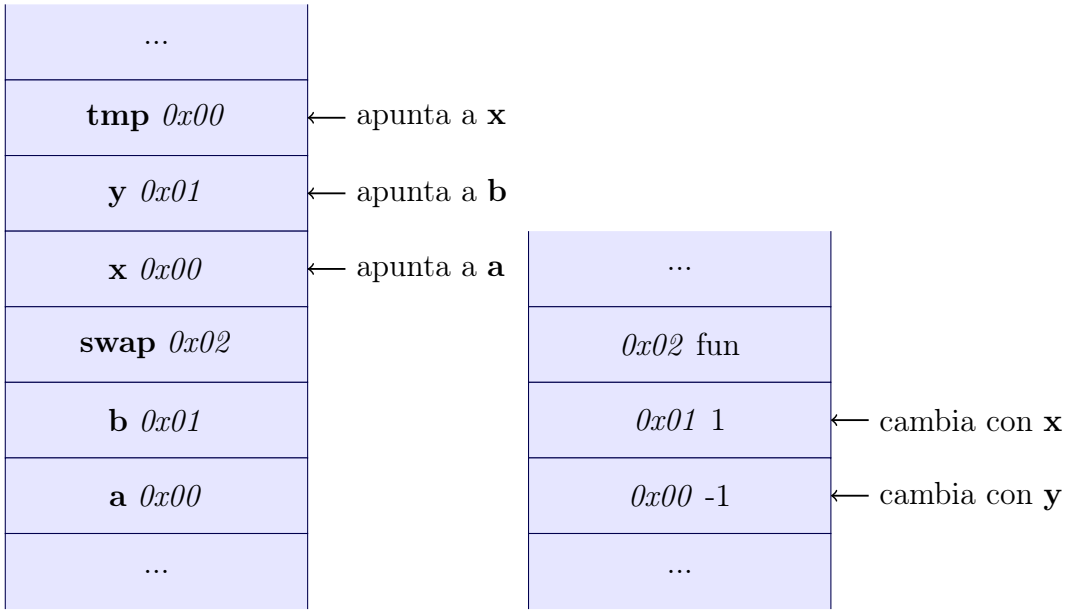
Se tiene que:

parámetros reales: a b
parámetros formales: x y

```
{swap a b}

{fun {x y}
    {with {tmp x}
            {seqn {set x y}
                {set y tmp}}}}
```




```
{seqn {swap a b}
      {- a b}}
```




```
> (- a b)
> (- -1 1)
> -2
```

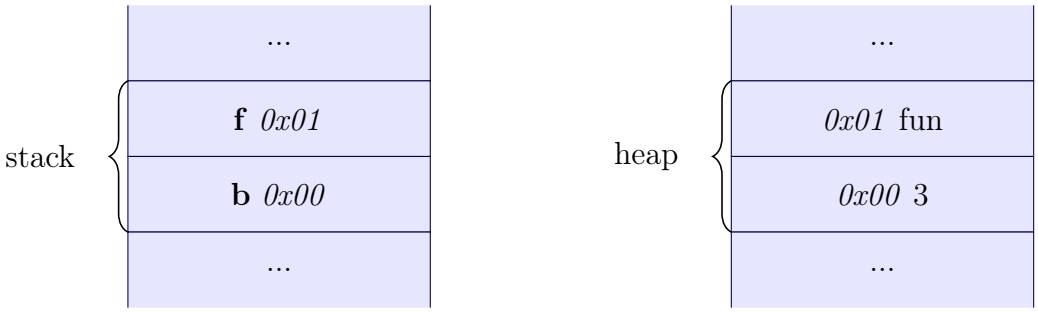
∴ el resultado en -2.

21. Evalúa la siguiente expresión usando paso de parámetros por necesidad.

```
{with {{b 3}
       {f {fun {x y}
            {seqn {set x 4}
                  y}}}}
  {+ {f b b} b}}
```

Se inicializan el *Stack* y el *Heap*:

```
{b 3}
  {f {fun {x y}
        {seqn {set x 4}
              y}}}
```

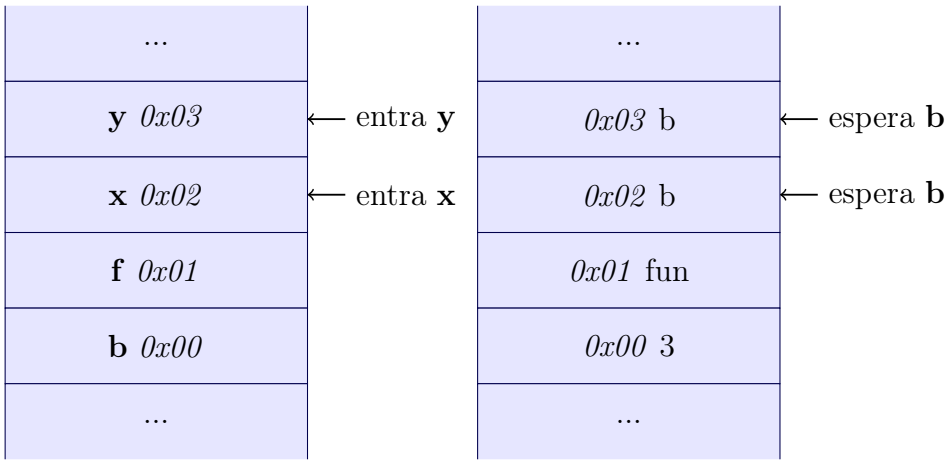


Se tiene que:

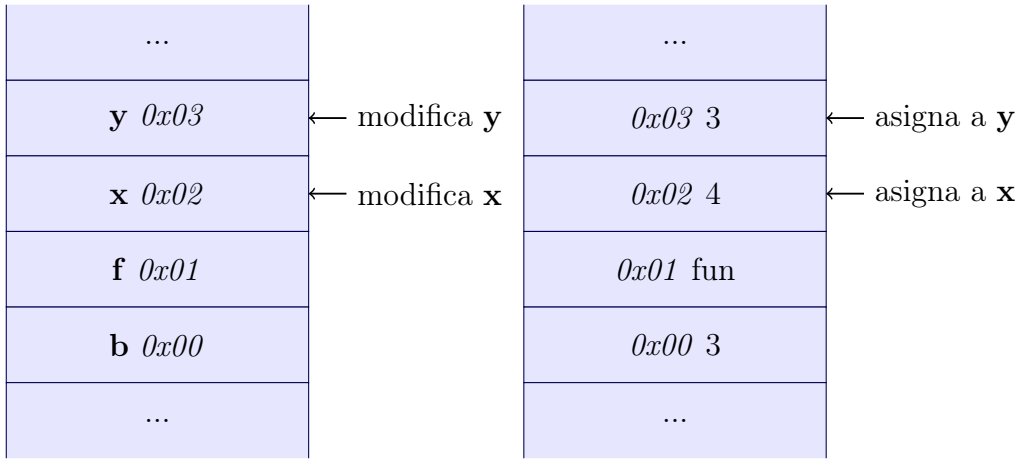
parámetros reales: b b

parámetros formales: x y

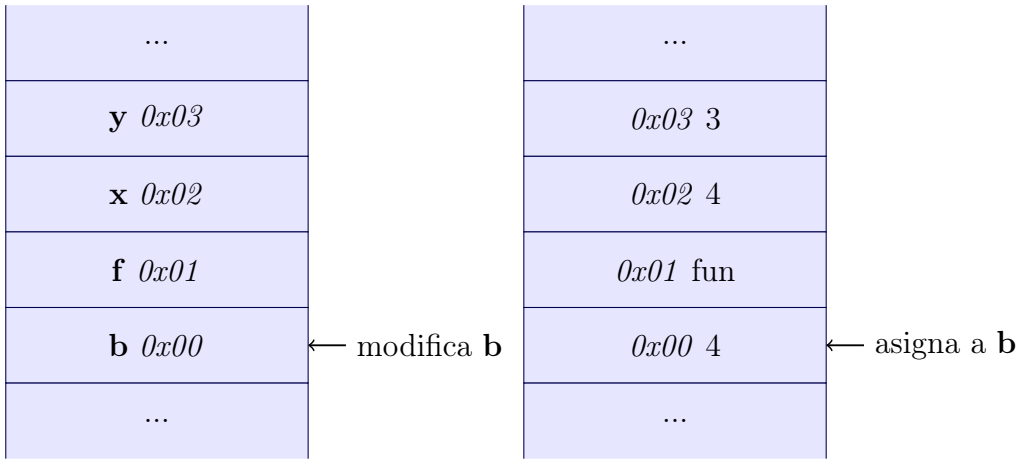
```
{+ {f b b} b}
```



```
{with {{b 3}
       {f {fun {x y}
            {seqn {set x 4}
                  y}}}}
  {+ {f b b} b}}
```




```
> (+ (f b b) b)
> (+ (f 3 3) b)
> (+ 3 b)
```




```
> (+ (f b b) b)
> (+ (f 3 3) b)
> (+ 3 b)
> (+ 3 4)
> 7
```

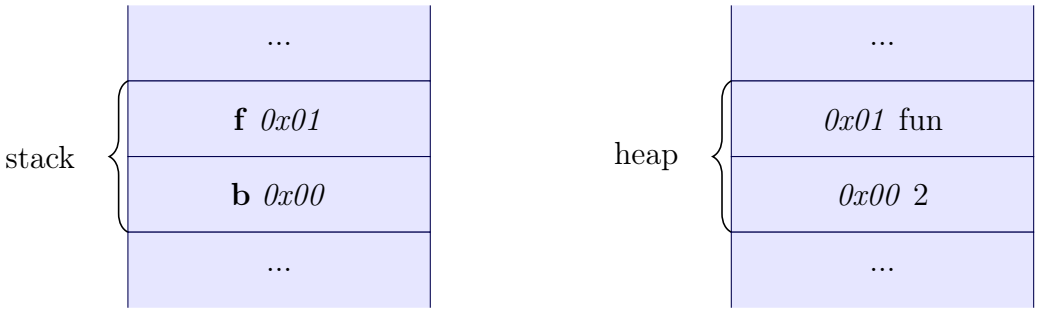
$\therefore$ el resultado es 7.

22. Evalúa la siguiente expresión usando paso de parámetros referencia-regreso.

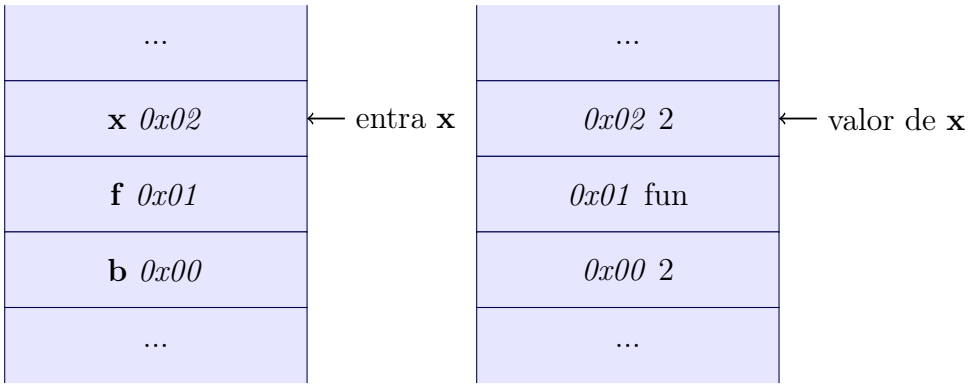
```
{with {{b 2}
       {f {fun {x}
            {seqn {set x 4}
                  {+ x b}}}}}
  {+ {f b} b}}
```

Se inicializan el *Stack* y el *Heap*:

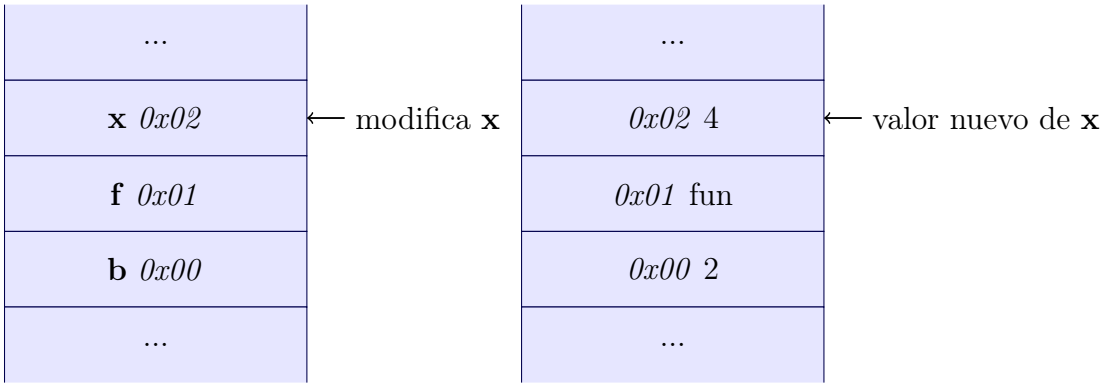
```
{b 2}
   {f {fun {x}
        {seqn {set x 4}
              {+ x b}}}}
```



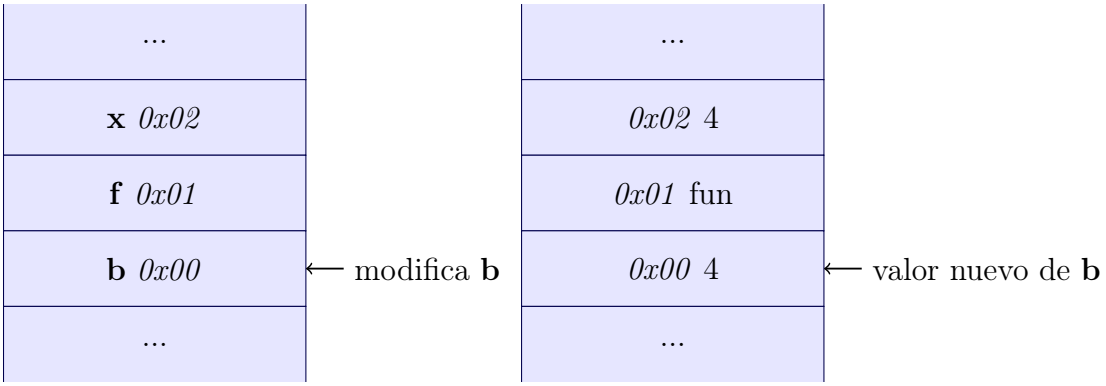

```
{+ {f b} b}
```



```
{fun {x}
     {seqn {set x 4}
     {+ x b}}}
```




```
> (+ (f b) b)
> (+ (f 2) b)
> (+ 6 b)
```




```
> (+ (f b) b)
> (+ (f 2) b)
> (+ 6 b)
> (+ 6 4)
> 10
```

$\therefore$ el resultado es 10.

23. Define los siguientes conceptos y pon ejemplos relacionados con cada uno:

    a. Tipo (tipos básicos, Tipos Abstractos de Datos, en particular con los tipos básicos da una categorización de éstos)

    - Tipos Básicos

      Son aquellos tipos que ya vienen predeterminados de forma nativa en el lenguaje, tales como: int, bool, char, float, entre otros.

    - Tipos Abstractos

      Son todos aquellos datos que se encuentran de forma primitiva, tales como Árboles o cadenas.

    b. Lenguaje de propósito general

    Los lenguajes de propósito general son aquellos que son empleados para muchas cosas de la industria, esto es, que pueden utilizarse para realizar diversas tareas de distintas áreas.

    c. Lenguaje envolvente

    Los lenguajes envolventes son aquellos que permiten codificar en algún otro lenguaje dentro del mismo, es decir, que un lenguaje es capaz que almacenar y ejecutar código de otro lenguaje diferente.

    d. Lenguajes con tipificado estático.

    Son aquellos lenguajes en los cuales sus tipos son definidos en tiempo de compilación.

24. Explica a profundidad las propiedades de (1) consistencia y (2) seguridad, da ejemplos de ambos.

    a. Consistencia

    La consistencia es la que nos asegura que el tipo definido de cualquier variable, se preservará a lo largo de la ejecución de un programa, esto es que el tipo de variable no se modificará hasta que devuelva un resultado con el mismo tipo con el que se definió la variable o variables involucradas,por ejemplo, si se declaran dos booleanos p y q la operación *or* devuelve un booleano de nuevo.

    b. Seguridad

    La propiedad de seguridad es la que nos asegura que ninguna de las operaciones primitivas del lenguaje, se aplique a los tipos incorrectos. Cada una de estas solo puede operar con ciertos tipos, pero si por alguna razón se quiere operar con tipos incorrectos, la seguridad se encarga de frenar esto. Por ejemplo, (* 5 9) que claramente ambos parámetros son números; (* 2 #f) no puede realizarse y la seguridad se encarga de lanzar un error.

25. Menciona 3 ventajas que presentan los lenguajes tipificados implícitamente sobre los tipificados explícitamente.

- Requiere menos esfuerzo pues el intérprete realiza conversiones
- Poseen mayor portabilidad.
- Son mucho más flexibles.

26. Menciona si son falsas (F) o verdaderas (V) cada uno de las afirmaciones sobre el sistema verificador de tipos.

a. ***V*** El sistema verificador de tipos no rechaza algunos programas eficientes.

b. ***V*** El sistema verificador de tipos ayuda a reducir el tiempo de depuración de un programa.

c. ***F*** El sistema verificador de tipos rechaza algunos programas válidos.

d. ***F*** En tiempo de ejecución nunca se verifican los tipos de un programa.

e. ***V*** El sistema verificador de tipos detecta errores en programas que hayan pasado el análisis sintáctico.

f. ***V*** El sistema verificador de tipos cacha todos los errores en tiempo de compilación.

27. Realiza el juicio de tipo para las siguientes expresiones:

a. `{rec {fib {<implementa fibonacci>}} {fib 2}}`

```
{rec {fib:(number -> number)
        {fun {n:number}:number
              {if {zero? n}
                   0
                   {if {one? n}
                       1
                       {+ {fib {- n 1}} {fib {- n 2}}}}}}}}
      {fib 2}}
```

El jucio se encuentra el la siguiente liga y también en *src/image/jucio1.png*.

b. `{{fun {x:number}:number`
`{+ {* x y } {+ 2 2}}}`
`8}`

$$\dfrac{\dfrac{\Gamma \vdash \texttt{x:number} \quad \Gamma \vdash \texttt{y:number}}{\Gamma \vdash \texttt{\{* x y\}:number}} \quad \dfrac{\varnothing \vdash \texttt{2:number} \quad \varnothing \vdash \texttt{2:number}}{\varnothing \vdash \texttt{\{+ 2 2\}:number}}}{\dfrac{\varnothing \vdash \texttt{\{fun \{x:number\}:number \{+ \{* x y\}\} \{+ 2 2\}\}:number} \quad \varnothing \vdash \texttt{8:number}}{\varnothing \vdash \texttt{\{\{fun \{x:number\}:number \{+ \{* x y\}\} \{+ 2 2\}\} 8\}:number}}}$$

donde $\Gamma = [x \leftarrow number, y \leftarrow number]$

28. De la siguiente expresión, da la inferencia de tipos y menciona de qué tipo son las variables al final de la inferencia realizada, asígnale un nombre mnemotécnico a la función.

a.
```
(define (mystery lis)
  (cond
    [(empty? lis) 0]
    [else (+ 1 (mystery (cdr lis)))]))
```

Primero se obtienen las subexpresiones:

1. `(cond [(empty? lis) 0] [else (+ 1 (mystery (cdr lis)))])`
2. `(empty? lis)`
3. `0`
4. `(+ 1 (mystery (cdr lis)))`
5. `(mystery (cdr lis))`
6. `(cdr lis)`

Ahora inferimos los tipos:

$[\![1]\!] = [\![\texttt{(cond [(empty? lis) 0] [else (+ 1 (mystery (cdr lis)))])}]\!]$

$= [\![2]\!]$ regresa $[\![3]\!]$ o regresa $[\![4]\!]$

$[\![2]\!] = [\![\texttt{(empty? lis)}]\!] = \texttt{boolean}$

$= [\![\texttt{lis}]\!] \rightarrow \texttt{boolean}$ y $[\![\texttt{lis}]\!] = \texttt{list}$

$[\![3]\!] = [\![0]\!] = \texttt{number}$

$[\![4]\!] = [\![\texttt{(+ 1 (mystery (cdr lis)))}]\!] = \texttt{number}$

$= [\![1]\!] = \texttt{number}$ y $[\![\texttt{(mystery (cdr lis))}]\!] = [\![5]\!] = \texttt{number}$

$[\![\boxed{6}]\!] = [\![\texttt{(cdr lis)}]\!] = \texttt{list}$ y $[\![\texttt{lis}]\!] = \texttt{list}$

$\therefore [\![\texttt{mystery}]\!] = \texttt{list} \rightarrow \texttt{number}$.

Ahora, una mejor forma de llamar a la función `mystery`:

```
1
2  ;; Obtiene la longitud de una lista
3  ;; length-list :: (listof any) --> number
4  (define (length-list lis)
5    (cond
6      [(empty? lis) 0]
7      [else (+ 1 (length-list (cdr lis)))]))
8
```

29. Utiliza el algoritmo de unificación visto en clase con la expresión

$$\texttt{((lambda (x) x) (* 7 1))}$$

Primero se obtienen las subexpresiones:

$\boxed{1}$ `((lambda (x) x) (* 7 1))`

$\boxed{2}$ `(lambda (x) x)`

$\boxed{3}$ `(* 7 1)`

$\boxed{4}$ `7`

$\boxed{5}$ `1`

Ahora, el proceso de unificación:

Paso	**Stack**	**Sustitución**
inicio	$[\![2]\!] = [\![3]\!] \to [\![1]\!]$ $[\![2]\!] = [\![x]\!] \to [\![x]\!]$ $[\![3]\!] = \texttt{number}$ $[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	empty
2	$[\![3]\!] \to [\![1]\!] = [\![x]\!] \to [\![x]\!]$ $[\![3]\!] = \texttt{number}$ $[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto [\![3]\!] \to [\![1]\!]$
4	$[\![3]\!] = [\![x]\!]$ $[\![1]\!] = [\![x]\!]$ $[\![3]\!] = \texttt{number}$ $[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto [\![3]\!] \to [\![1]\!]$
2	$[\![1]\!] = [\![x]\!]$ $[\![x]\!] = \texttt{number}$ $[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto [\![3]\!] \to [\![1]\!]$ $[\![3]\!] \mapsto [\![x]\!]$
2	$[\![x]\!] = \texttt{number}$ $[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto [\![3]\!] \to [\![1]\!]$ $[\![3]\!] \mapsto [\![x]\!]$ $[\![1]\!] \mapsto [\![x]\!]$
2	$[\![4]\!] = \texttt{number}$ $[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto \texttt{number} \to \texttt{number}$ $[\![3]\!] \mapsto \texttt{number}$ $[\![1]\!] \mapsto \texttt{number}$ $[\![x]\!] \mapsto \texttt{number}$
2	$[\![5]\!] = \texttt{number}$	$[\![2]\!] \mapsto \texttt{number} \to \texttt{number}$ $[\![3]\!] \mapsto \texttt{number}$ $[\![1]\!] \mapsto \texttt{number}$ $[\![x]\!] \mapsto \texttt{number}$ $[\![4]\!] \mapsto \texttt{number}$
2	empty	$[\![2]\!] \mapsto \texttt{number} \to \texttt{number}$ $[\![3]\!] \mapsto \texttt{number}$ $[\![1]\!] \mapsto \texttt{number}$ $[\![x]\!] \mapsto \texttt{number}$ $[\![4]\!] \mapsto \texttt{number}$ $[\![5]\!] \mapsto \texttt{number}$

30. Da tres ventajas de contar con polimorfismo explícito en un lenguaje de programación. Utiliza ejemplos.

a. Permite identificar por medio de los tipos que recibe, por ejemplo:

```
public void volar(Vehiculo avion) {...}
public void volar(Animal paloma) {...}
```

b. Permite la sobrecarga de métodos, por ejemplo:

```
public int suma(int a, int b) {
      return a + b;
}

public int suma(int a, int b,int c) {
      return a + b + c ;
}
```

c. Permite definir un mismo método para distintos tipos de datos, por ejemplo:

```
public concatena(String str1, String str2) {
      return str1 + str2;
}

public List<Boolean> suma(List<Boolean> a, List<Boolean> b) {
      a = new ArrayList<Boolean>();
      return  a.addAll(b);
}
```

31. Explica los conceptos y las diferencias que existen entre el uso de macros y de funciones en un lenguaje de programación.

- Funciones

  Las funciones son instrucciones independientes que realizan una tarea específica.

- Macros

  Las macros, en pocas palabras, son pedazos de código que el compilador coloca en el sitio de invoación.

Algunas veces, pueden llegar a confundirse ya que parece que hacen lo mismo, por lo que algunas diferencias entre ambas:

- Las funciones son incapaces de controlar lo que pueda llegar a ocurrir con sus parámetros, es decir, no puede asegurar si se evalúa o no mientras que las macros pueden hacerlo sin problema.

32. Dentro del paradigma de los lenguajes orientados a objetos, da un programa donde pongas en práctica los conceptos de objetos, clases, herencia y polimorfismo explícito. Explica cada uno de los conceptos y su uso en tu programa (no usar pseudo-código).

Clase Madre

```
import java.text.SimpleDateFormat;
import java.util.Date;
/**
 * Clase que simula el comportamiento de un boleto
 * @author Ramses Lopez
 */
 public class Boleto{
   /*nombre del evento*/
   protected String evento;
   /*fecha del evento*/
   protected String fecha;
   /*precio del boleto*/
   protected double precio;
   /*numero de boleto*/
   protected String boleto;
   /*lugar del evento*/
   protected String ubicacion;

   /**
    * Constructor de un boleto vacio
    */
    public Boleto(){
      evento = "Unknown";
      precio = 0;
      boleto = "000";
      ubicacion = "Unknown";
      fecha = new SimpleDateFormat("dd-MM-yyyy HH:mm").format(new Date());
    }

    /**
     * Constructor de un nuevo boleto;
     * @param evento2 - evento que se va a realizar
     * @param precio2 - precio del boleto
     * @param boleto2 - tipo de boleto
     * @param ubicacion2 - lugar del evento
     * @param fecha2 - fecha del evento
     */
     public Boleto(String evento2, double precio2, String boleto2,
                   String ubicacion2, String fecha2){
       evento = evento2.equals(" ") ? "Unknown" : evento2;
       precio = precio2 == 0.0 ? 0.0 : precio2;
       boleto = boleto2.equals(" ") ? "000" : boleto2;
       ubicacion = ubicacion2.equals(" ") ? "Unknown" : ubicacion2;
       fecha = fecha2.equals(" ") ?
               new SimpleDateFormat("dd-MM-yyyy HH:mm").format(new Date())
               : fecha2;
     }


   /**
```

```
51      * Metodo para conocer el nombre del evento
52      * @return String - nombre del evento
53      */
54     public String obtenerEvento(){
55       return evento;
56     }
57
58     /**
59      * Metodo para asignar un evento
60      * @param newEvento - evento a asignar
61      */
62     public void asignarEvento(String newEvento){
63       evento = newEvento;
64     }
65
66     /**
67      * Metodo para conocer la fecha del evento
68      + @return String - fecha del eento
69      */
70     public String obtenerFecha(){
71       return fecha;
72     }
73
74     /**
75      * Metodo para asignar una fecha para el evento
76      * @param newFecha - fecha a asignar
77      */
78     public void asignarFecha(String newFecha){
79       fecha = newFecha;
80     }
81
82     /**
83      * Metodo para conocer el precio del boleto
84      * @return double - precio del evento
85      */
86     public double obtenerPrecio(){
87       return precio;
88     }
89
90     /**
91      * Metodo para asignar un precio
92      * @param newPrecio - precio a asignar
93      */
94     public void asignarPrecio(double newPrecio){
95       precio = newPrecio;
96     }
97
98     /**
99      * Metodo para conocer el tipo de boleto
100     * @return String - tipo de boleto
101     */
102    public String obtenerBoleto(){
103      return boleto;
104    }
105
106    /**
```

```
107     * Metodo para asignar el tipo de boleto
108     * @param newBoleto - tipo de boleto a asignar
109     */
110    public void asignarBoleto(String newBoleto){
111      boleto = newBoleto;
112    }
113
114    /**
115     * Metodo para conocer el lugar del evento
116     * @return String - lugar del evento
117     */
118    public String obtenerUbicacion(){
119      return ubicacion;
120    }
121
122    /**
123     * Metodo para asignar el lugar del evento
124     * @param newUbicacion - ubicacion a asignar
125     */
126     public void asignarUbicacion(String newUbicacion){
127       ubicacion = newUbicacion;
128     }
129
130     protected boolean comprarBoleto(int pago) {
131       if (pago >= 500) {
132           System.out.println("Compra exitosa");
133           return true;
134       } else {
135         System.out.println("Saldo insuficiente");
136         return false;
137       }
138     }
139
140    /**
141     * Metodo para conocer el tipo de boleto, hora
142     * precio, evento y fecha
143     * @return String - boleto
144     */
145    public String toString(){
146      //return " Evento: " + evento + "\n Boleto: " + boleto + "\n Precio:
    " + precio +  "\n Fecha: " + fecha + "\n Lugar: " + ubicacion;
147      return "+----------------------------+ \n"
148           + "| Evento: " + evento +      "| \n"
149           + "| Boleto: " + boleto +      "| \n"
150           + "| Precio: " + precio +      "| \n"
151           + "| Fecha: " + fecha +        "| \n"
152           + "| Lugar: " + ubicacion +    "| \n"
153           + "+----------------------------+";
154    }
155  }
```

Clase Hija

```java
/**
 * Clase que simula un boleto de estudiante
 * @see Boleto - clase boleto
 * @author Ramses Lopez Soto
 */
public class BoletoEstudiante extends Boleto {

  int[] precios = {500, 750 , 1000};

  /**
   * Consructor por omision
   */
   public BoletoEstudiante(){
     super();
   }

  /**
   * Constructor para hacer descuento si se es estudiante de
   * de la UNAM, IPN o UAM
   * @param evento - evento al que se va a asistir
   * @param precio - precio del boleto
   * @param boleto - tipo de boleto
   * @param lugar - lugar del evento
   * @param fecha - fecha del evento
   * @param escuela - escuela de procedencia
   */
  public BoletoEstudiante(String evento, double precio, String boleto,
  String lugar, String fecha, String escuela){
    super(evento, precio, boleto, lugar, fecha);
    if (validarInstitucion(escuela) == true) {
      System.out.println("Tienes el 50% de descuento en tu boleto");
    } else{
      System.out.println("El costo de tu boleto es regular");
    }
  }

  /**
   * Metodo para verificar la pertenecia a la UNAM, IPN o UAM
   * @param institucion - escuela de procedencia
   * @return boolean - true si pertenece y false en otro caso
   */
  public boolean validarInstitucion(String institucion){
    if(institucion.equalsIgnoreCase("UNAM") || institucion.
  equalsIgnoreCase("IPN") || institucion.equalsIgnoreCase("UAM")){
      return true;
    } else {
      return false;
    }
  }

  public int[] comprarBoleto() {
    System.out.println("Los precios dispodibles son");
    return precios;
  }

```

```
54    /**
55     * Metodo para obtener el boleto impreso
56     * @return String - boleto de estudiante
57     */
58     public String toString(){
59        return "+----------------------------+ \n"
60              + "| Evento: " + evento +      "| \n"
61              + "| Boleto: BE - " + boleto + "| \n"
62              + "| Precio: " + precio +      "| \n"
63              + "| Fecha: " + fecha +        "| \n"
64              + "| Lugar: " + ubicacion +    "| \n"
65              + "+----------------------------+";
66     }
67  }
```

Los programas anteriores poseen:

- Objetos

```
protected String fecha;
fecha = new SimpleDateFormat("dd-MM-yyyy HH:mm").format(new Date());
```

- Clases

```
public class Boleto{ ... }
```

- Herencia

```
public class BoletoEstudiante extends Boleto { ... }
public BoletoEstudiante(){
      super();
}
```

- Polimorfismo (explicito)

```
protected boolean comprarBoleto(int pago) { ... }
public int[] comprarBoleto() { ... }
```

⋆ FIN ⋆

†🦇†