Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

•••

Estado

Karla Ramírez Pulido

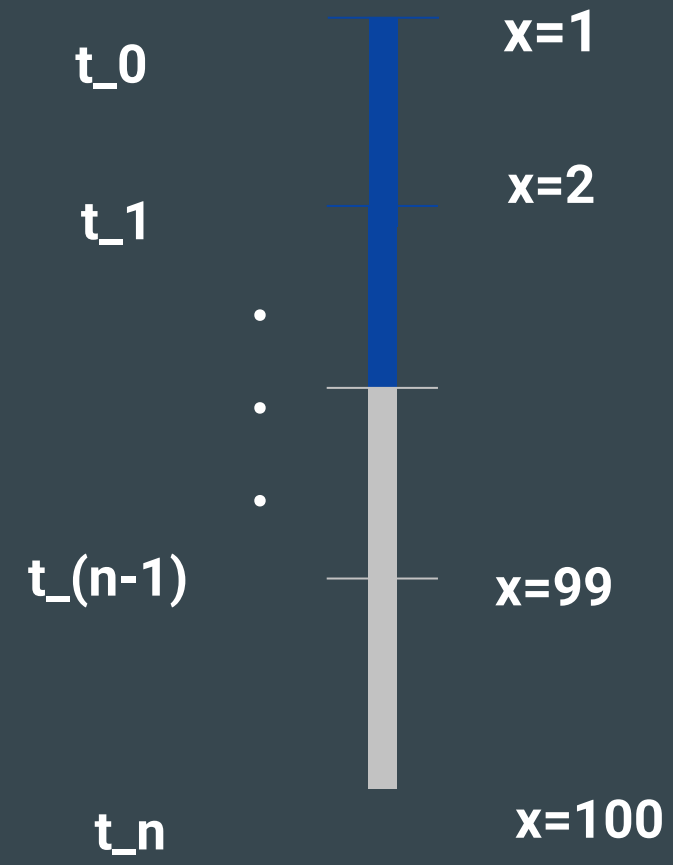
Idea inicial

Supongamos que tenemos una variable inicializada con el valor de 1:

x = 1

En la ejecución (t-1, t_2, t_3, t_4, t_5, … , t_(n-1)

Termina con un valor que puede ser diferente en el tiempo t_n
Por ejemplo:
X= 100



Estado

Mutación o cambio de valores asociados a través de “nombres”.

En Racket: simulamos el estado usando CAJAS.

En Haskell NO hay operaciones de cambio de estado.

Gramática

<BCFAE> ::= ...

| {newbox <BCFAE>}

| {setbox <BCFAE> <BCFAE>}

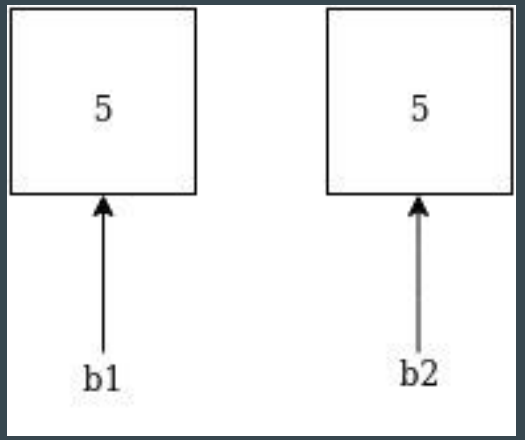
| {openbox <BCFAE>}

| {seqn <BCFAE> <BCFAE>}

Ejemplos 1 de uso con CAJAS

(define b1 (box 5))

(define b2 (box 5))



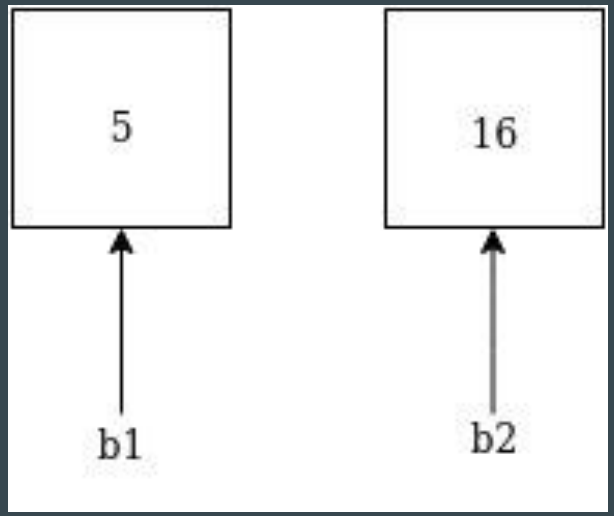
(set-box! b2 16)

(unbox b2)

16

(unbox b1)

5



Diferencias entre gramática nuestro MINI-interp y Racket

```
{with  {b  {newbox  0}}
      {seqn
            {setbox b {+ 1 {openbox b}}}
            {openbox b}}
```

```
(local ([ define b (box 0)])
      (begin
            (set-box! b (+ 1 (unbox  b)))
            (unbox  b)))
```

Operación: seqn

```
[seqn (e1 e2)
    (begin (interp e1 env)
        (interp e2 env))]
```

NOTA:

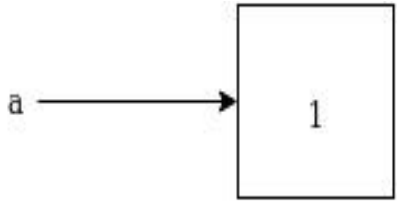
begin: interpreta línea por línea pero solo regresa el resultado de haber evaluado la última línea.

seqn: sólo podrá recibir 2 expresiones, evalúa la primera, pero solo regresa el resultado de haber evaluado la segunda. Sin embargo SI hace la evaluación de la primera expresión.

Ejemplo de seqn

```
{with {a {newbox 1}}
    {seqn
        {with {b 3}
            b}
        b}}
```

¿Cuál es el resultado de evaluar la expresión anterior?



Vemos el **seqn**

1era expresión: {with {b 3} b}

2da expresión: b

___

```
{with {a {newbox 1}}
    {seqn
        {with {b 3}
            b}
        b}}
```

¿Cuál es el resultado de evaluar la expresión anterior?

seqn: en el with b 3, i.e. asigna a la variable de ligado b el valor de 3

Cuerpo del with dentro del seqn es b

entonces ese with debe de regresar el valor de b = 3

sin embargo como está dentro del seqn regresa la b

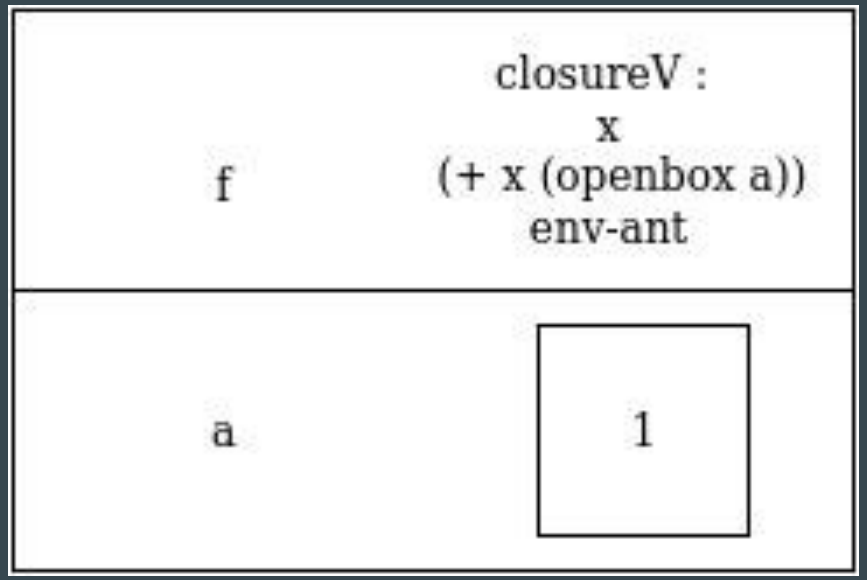
su valor es ¿?

ERROR se destruye el concepto de alcance estático

Ejemplo 2 de uso de cajas:

```
{with {a {newbox 1}}
      {with  {f  {fun {x}
                      {+  x  {openbox a}} }}
                 {seqn
                       {setbox a 2}
                       {f 5}}}}
```

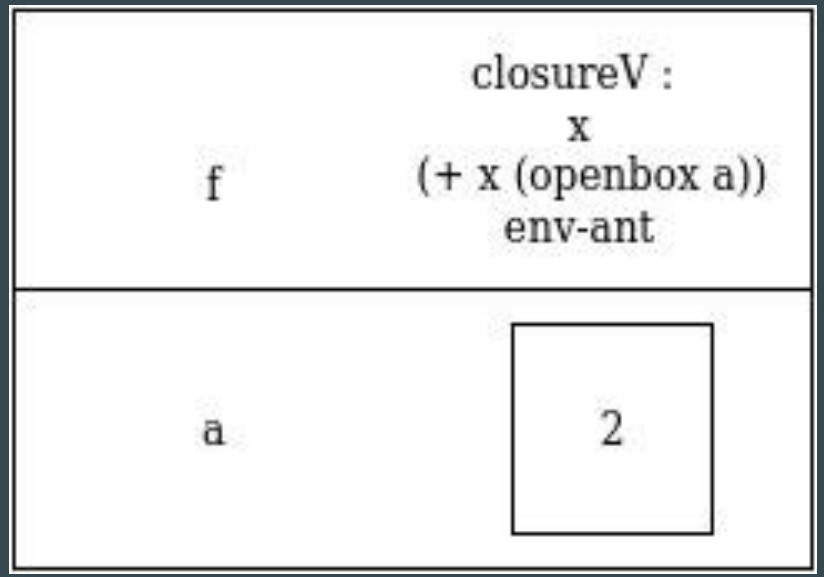
Ambiente



Ejemplo de uso de cajas:

```
{with {a {newbox 1}}
      {with  {f  {fun {x}
                      {+  x  {openbox a}} }}
                      {seqn
                            {setbox a 2}
                            {f 5}}}}
```

Ambiente



Ejemplo de uso de cajas:

{f 5}

{closureV: x, (+ x (openbox a),env-ant)}

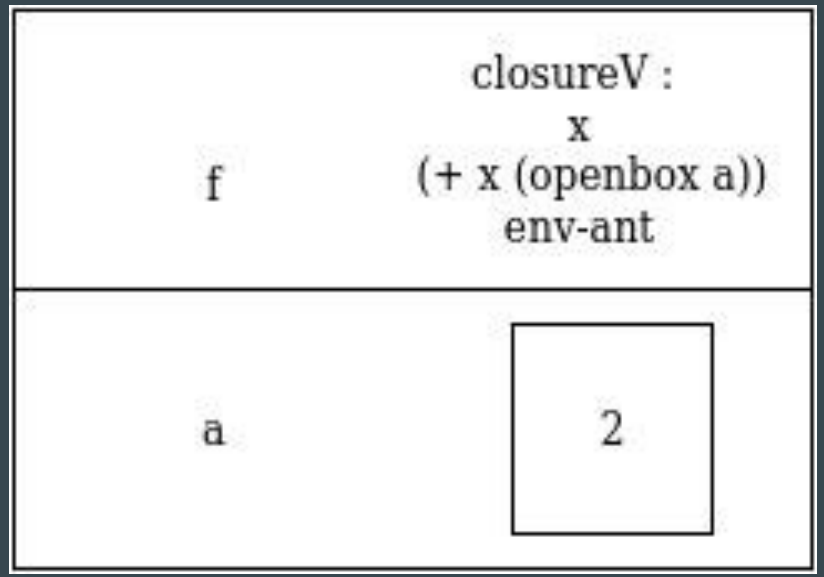
5}

Entonces la x=5

(+ 5 (openbox a))

(+ 5 2) = 7

Ambiente



Ejemplo 3

```
{with {x 3}
    {with {f  {fun {y} {+ x y}}}
        {with {x 5}
            {f 10}}}}
```

Ambiente:

x	5
f	closureV : y (+ x y) Amb-ant: (x 3)
x	3

Ejemplo 3

{f 10}

= {{fun {y} {+ x y}} 10}

y= 10

{+ x 10}

¿Cuál x toma?

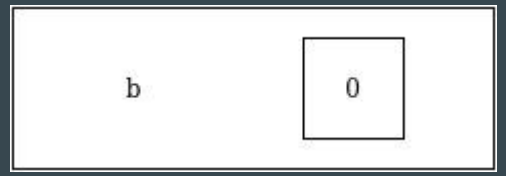
{+3 10} = 13

Ambiente:

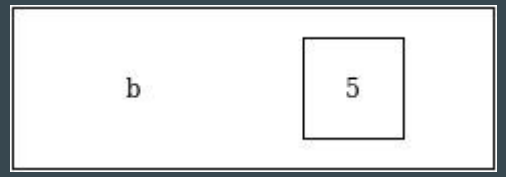
x	5
f	closureV : y (+ x y) Amb-ant: ((y 10), (x 3))
x	3

Ejemplo 4

```
{with {b {newbox  0}}
      {if0  {seqn
                  {setbox b 5}
                  {openbox b}}
            1
           {openbox b}}}
```

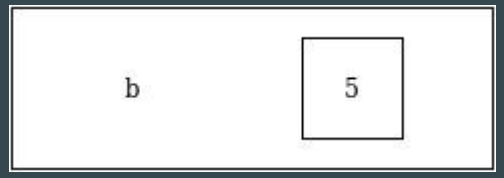


Esa misma b muta su valor y queda:



Ejemplo 4

```
{with {b {newbox  0}}
      {if0  {seqn
                  {setbox b 5}
                  {openbox b}}
            1
           {openbox b}}}
```



Sacamos lo que tiene b = 5

Evalúamos el (if0 5) → FALSO

Regresamos la evaluación de la rama del else, i.e.

{openbox b}

= 5

Store-Passing Style

Estilo de Paso de Almacenamiento

tenemos que cambiar la implementación

Store-Passing Style

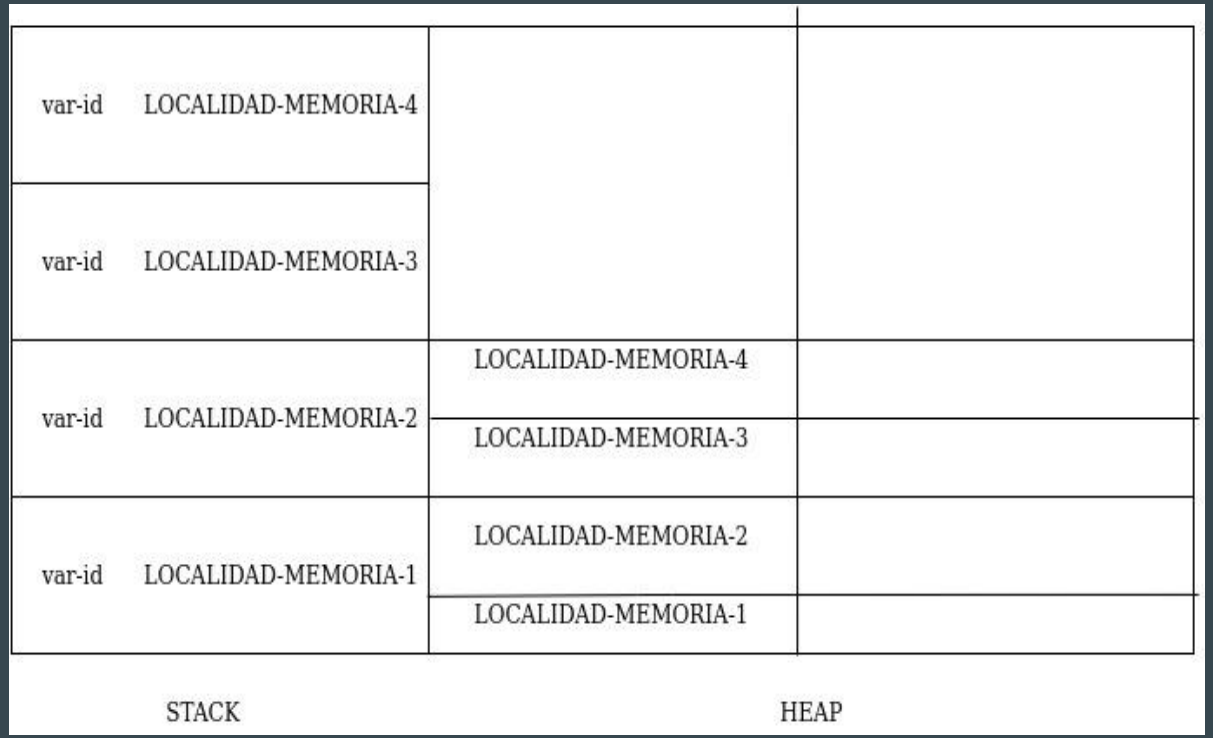
Ambiente:

- stack

  (var + loc-mem)

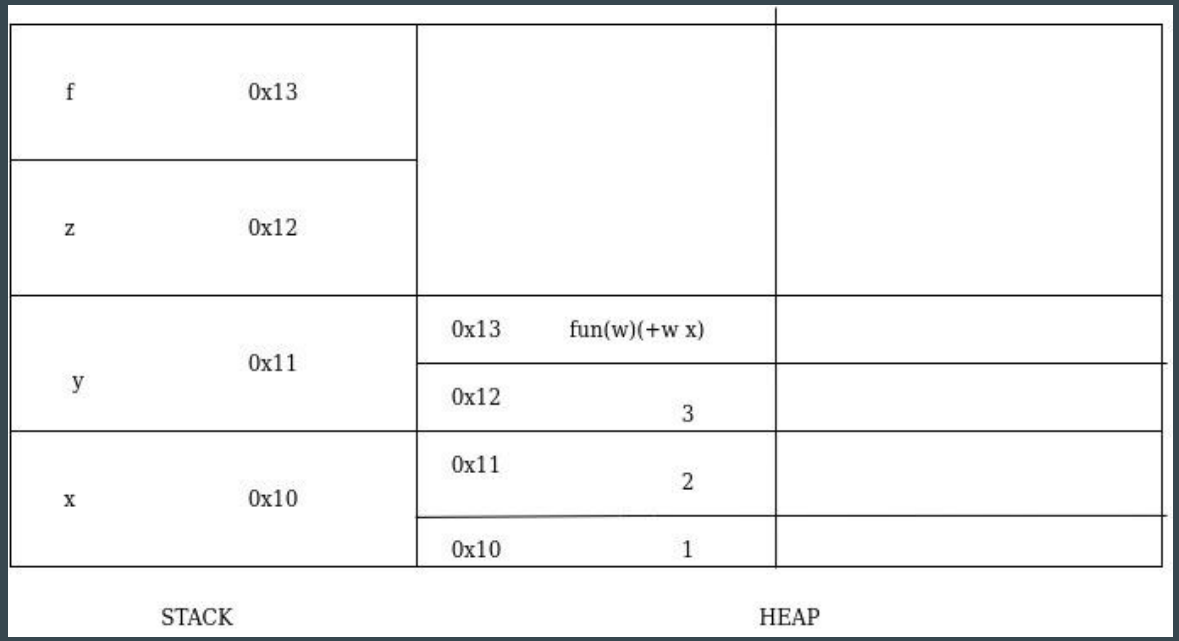
- heap

  (loc-mem valor)



Ejemplo 1 de Store-passing style

```
{with {x 1}
  {with {y 2}
    {with {z 3}
      {with {f {fun(w) (+ w x)}}
        {f 4}}}}}
```



Ejemplo 1 de Store-passing style

{f 4}

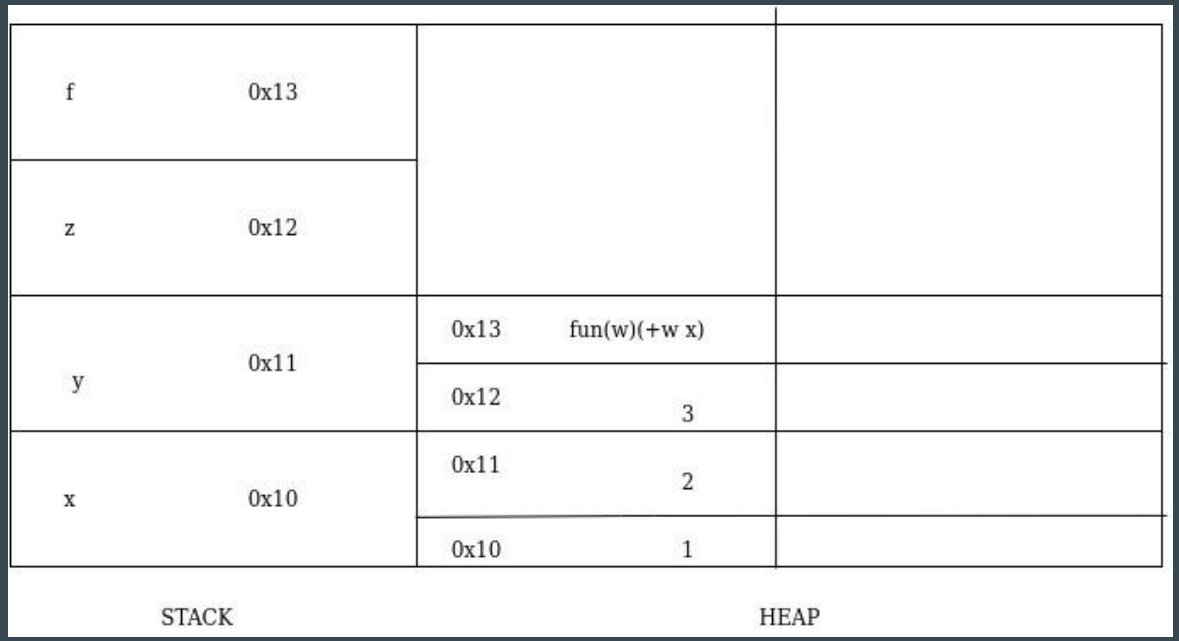
{ {fun (w) (+ w x)} 4}

w = 4

Evaluamos el cuerpo de la

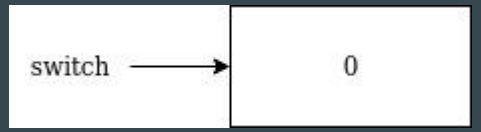
función: (+ w x) =

(+ 4 1) = 5



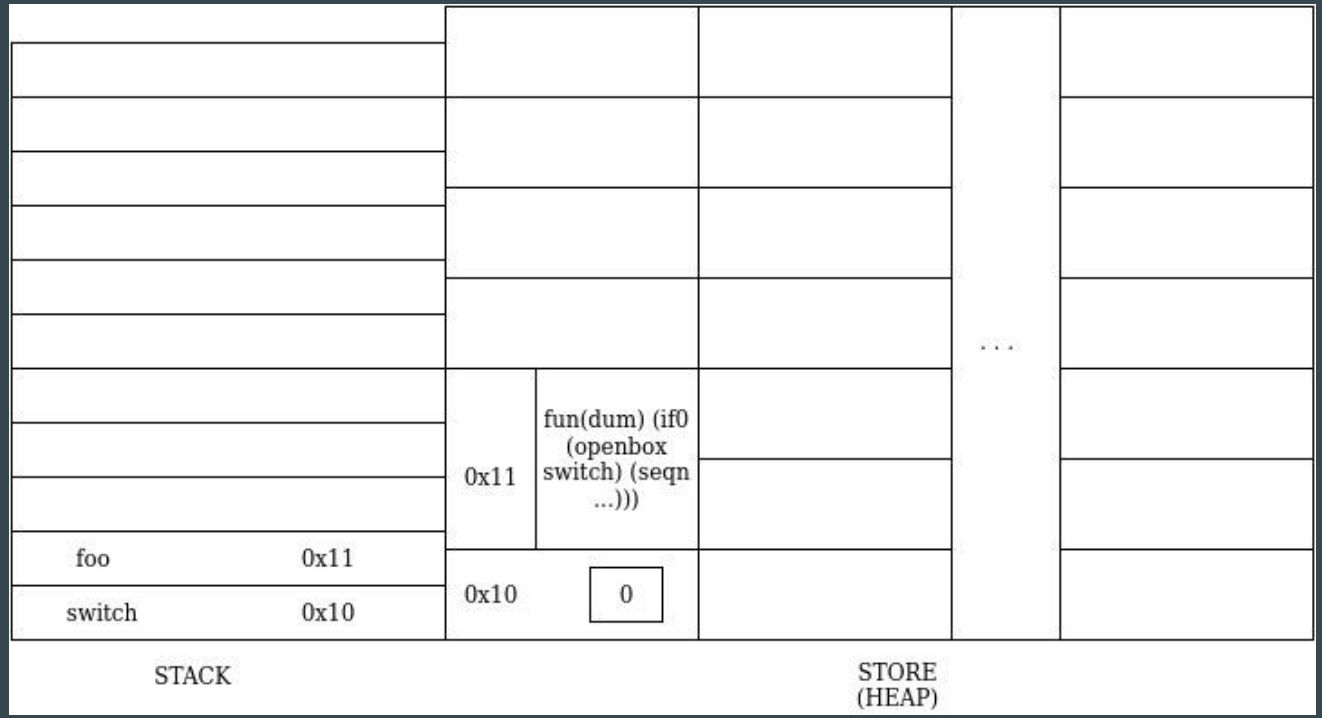
Ejemplo 2 de Store-Passing Style

```
{with {switch {newbox 0}}
    {with {foo {fun {dum}
                {if0 {openbox switch}
                     {seqn
                          {setbox switch 1}
                          1}
                     {seqn
                          {setbox switch 0}
                          0} }}}...}}
```



Ejemplo 2 de Store-Passing Style

```
{with {switch {newbox 0}}
  {with {foo {fun {dum}
    {if0 {openbox switch}
      {seqn
       {setbox switch 1}
       1}
      {seqn
       {setbox switch 0}
       0} }}}...}}
```



> (interp foo 1729)

```
{with {foo {fun {dum}
      {if0 {openbox switch}
           {seqn
           {setbox switch 1}
          1}
          {seqn
          {setbox switch 0}
          0} }}}
```

foo = (fun (dum) i.e. foo 1729
dum = 1729

Evaluamos if0 (openbox switch)

i.e. switch es 0

entonces evaluamos la rama del then

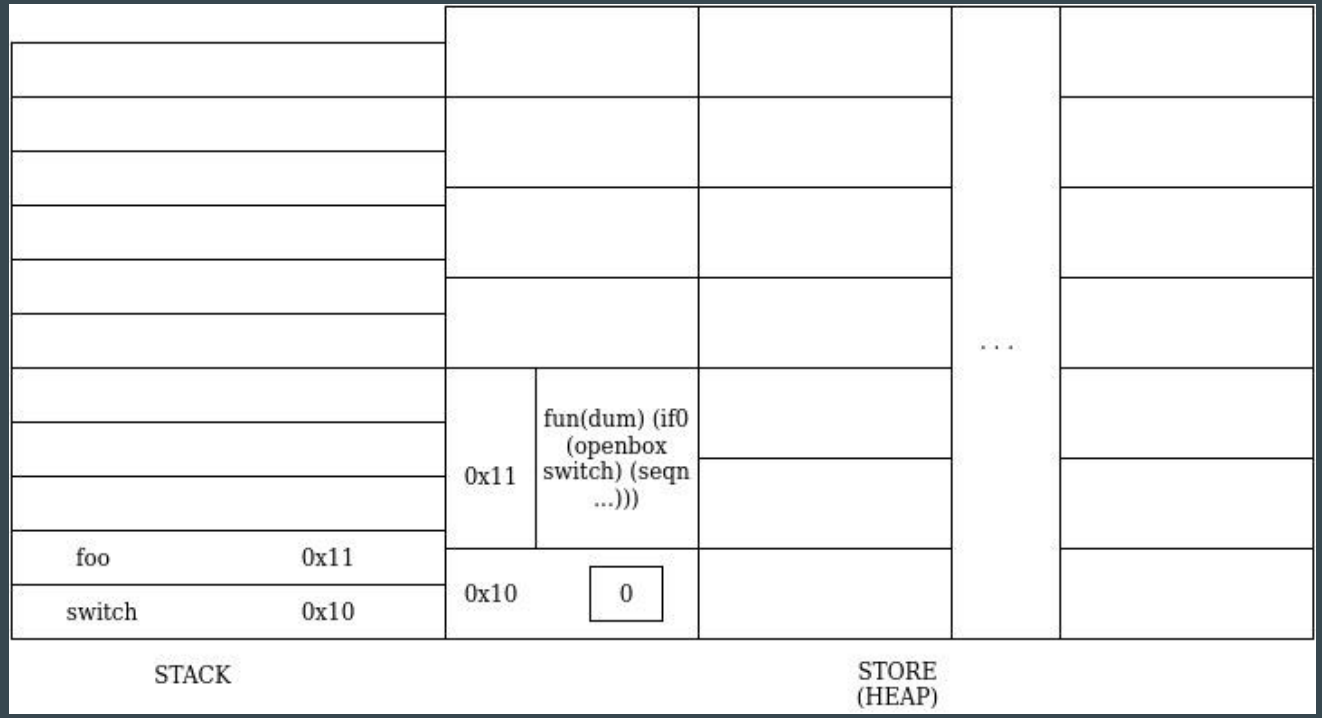
switch tiene el valor inicial de 0

Ambiente

en el cual

estamos

evaluando:



> (interp foo 1729)

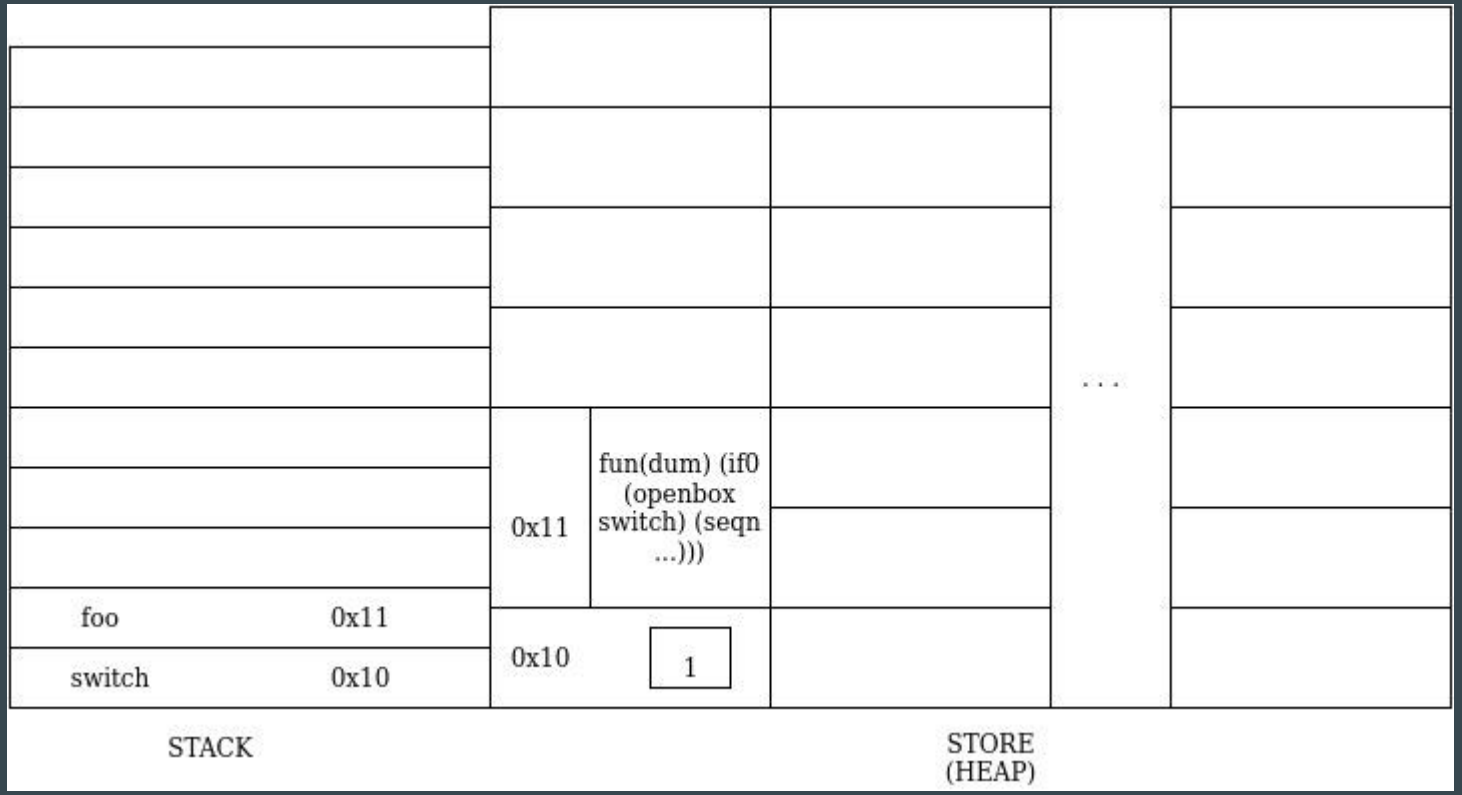
```
{with {foo {fun {dum}
      {if0 {openbox switch}
           {seqn
           {setbox switch 1}
          1}
          {seqn
          {setbox switch 0}
          0} }}}
```

entonces evaluamos la rama del then

asignar a switch = 1
 y regresamos un 1 (segunda línea del seqn)

switch tiene el valor de 1 (ya cambió su valor)

Ambiente en el cual estamos evaluando:



Entonces la llamada de (foo 1729)

Regresa el valor de 1

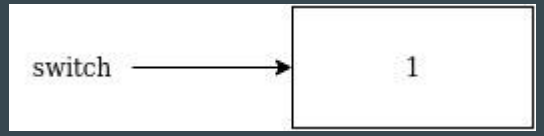
Sin embargo el **Ambiente YA cambió**, y si volvemos a llamar a la función foo con la misma instancia, i.e.

> (foo 1729)

¿Qué regresa?

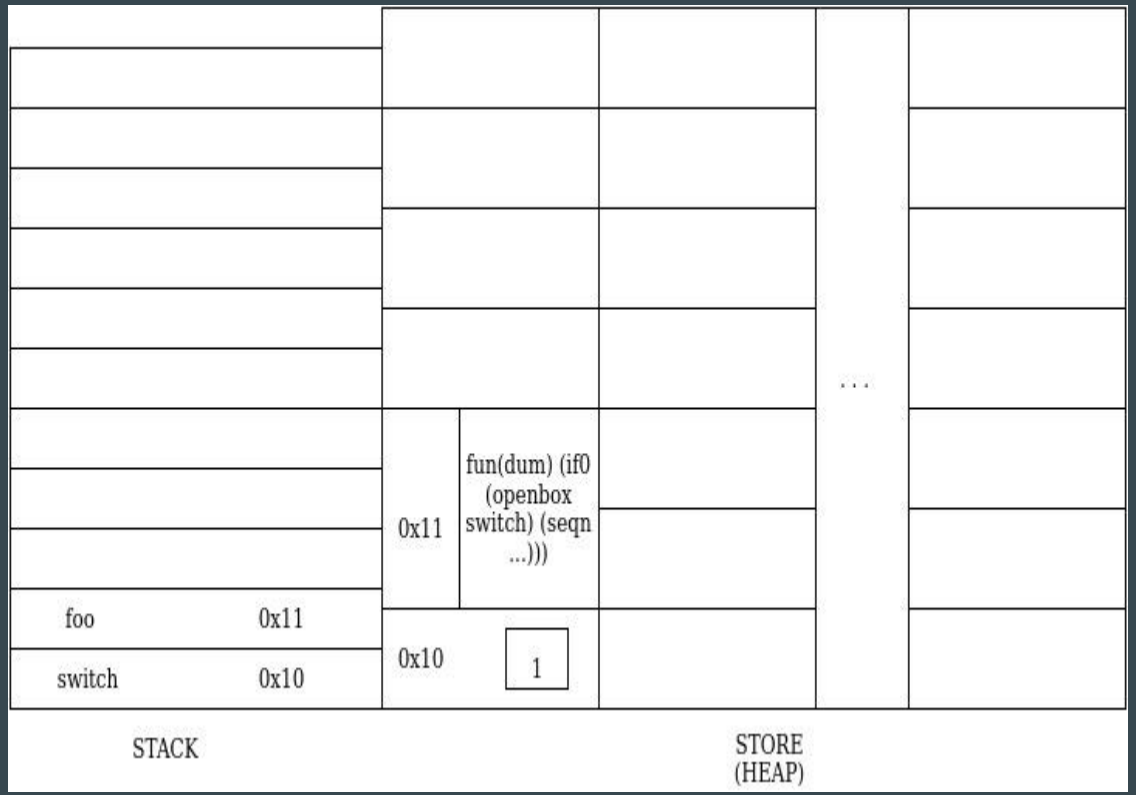
Ejemplo 2 de Store-Passing Style

```
{with {switch {newbox 0}}
    {with {foo {fun {dum}
                {if0 {openbox switch}
                     {seqn
                          {setbox switch 1}
                          1}
                     {seqn
                          {setbox switch 0}
                          0} }}}...}}
```



Ejemplo 2 de Store-Passing Style

```
{with {switch {newbox 0}}
  {with {foo {fun {dum}
    {if0 {openbox switch}
      {seqn
       {setbox switch 1}
       1}
      {seqn
       {setbox switch 0}
       0} }}}...}}
```



> (interp foo 1729)

```
{with {foo {fun {dum}
      {if0 {openbox switch}
           {seqn
           {setbox switch 1}
          1}
          {seqn
          {setbox switch 0}
          0} }}}
```

foo = (fun (dum) i.e. foo 1729
dum = 1729

Evaluamos if0 (openbox switch)

i.e. switch es 1

entonces evaluamos la rama del else

> (interp foo 1729)

```
{with {foo {fun {dum}
    {if0 {openbox switch}
        {seqn
        {setbox switch 1}
        1}
        {seqn
        {setbox switch 0}
        0} }}}
```

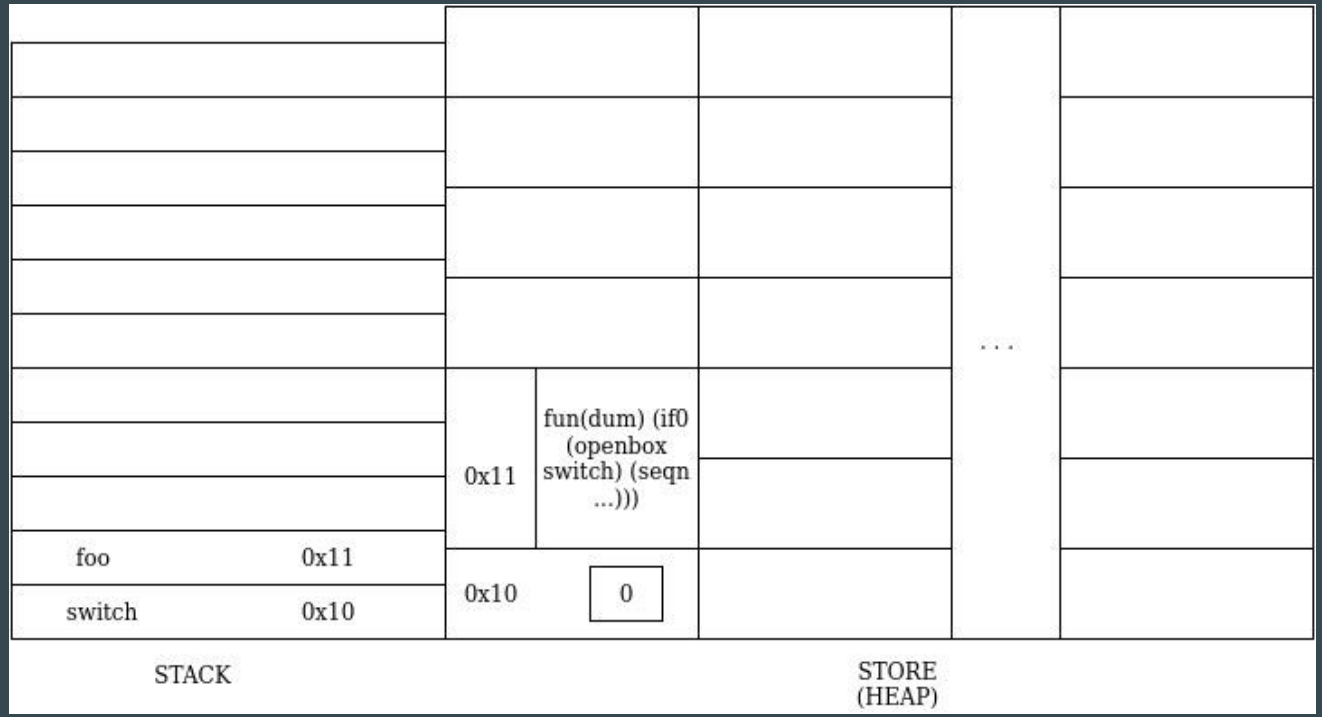
entonces evaluamos la rama del else

```
{seqn
  {setbox switch 0}
   0
}
```

Asignamos a **switch el valor de 0** y regresamos 0 (segunda línea del seqn anaranjado}

switch tiene el valor de 0

Ambiente en el cual estamos evaluando cambió otra vez su valor en switch:



La variable switch ¿cuántas veces mutó su valor?

Valor inicial en el tiempo t0 = 0

Valor en el tiempo t1 = 1

Valor en el tiempo t2 = 0

Implementación de Store-Passing Style

Datatype BCFAE-value

```
(define-type  BCFAE-Value
    [numV (n number?)]
    [closureV (param symbol?) (body BCFAE?) (env Env?)]
    [boxV (location number?)])
```

Implementación

```
(define-type  Value×Store
      [v×s (value BCFAE-Value?) (store Store?)])


;; interp:  BCFAE   Env   Store  →  Value×Store
(define (interp expr env store)
  [num (n) (v×s (numV n) store]
```

Implementación

```
;; interp:  BCFAE   Env   Store  →  Value×Store

(define (interp expr env store)

  [num (n) (v×s (numV n) store]

  [id (v)  (v×s (store-lookup (env-lookup v env) store) store)]

  [fun (bound-id  bound-body)  (v×s  (closureV  bound-id  bound-body  env)  store)]
```

Implementación

```
;; interp:  BCFAE   Env   Store  →  Value×Store

[if0 (test truth falsity)

     (type-case  Value×Store   (interp test env store)

          [v×s (test-value     test-store)

               (if (num-zero? test-value)

                    (interp truth  env  test-store)

                    (interp falsity   env  test-store))])]
```

Implementación

```
;; interp:  BCFAE   Env   Store  →  Value×Store

[add (l  r) (type-case Value×Store (interp  l  env  store)

    [v×s (l-value  l-store)

          (type-case  Value×Store  (interp  r   env  l-store)

                [v×s  (r-value  r-store)

                   (v×s  (num+   l-value   r-value)   r-store) ] ) ] ) ]
```

Implementación

```
;; interp:  BCFAE   Env   Store  →  Value×Store

[seqn (e1 e2)

      (type-case  Value×Store  (interp  e1  env  store)

            [v×s (e1-value   e1-store)

            (interp  e2   env  e1-store)])]
```

Implementación

```
;; interp:  BCFAE   Env   Store  →  Value×Store

[newbox (value-expr)

        (type-case  Value×Store  (interp  value-expr  env  store)

                [v×s (expr-value  expr-store)

                        (local  ([ define  new-loc  (next-location  expr-store)])

                                (v×s  (boxV  new-loc) (aSto  new-loc  expr-value  expr-store)))])]
```

En la sesión de ayudantía

Alan y Eliseo verá los casos siguientes y ejemplos:

- aplicación de función
- setbox
- openbox
- next-location

Gracias

¿Dudas?