



COMPILADORES 2020

COMPILANDO FUNCIONES

Basado en transparencias de X. Leroy

FUNCIONES EN FD4

FUNCIONES EN FD4

- ▶ FD4 tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumento
 - ▶ Puede devolver funciones como resultado

FUNCIONES EN FD4

- ▶ FD4 tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumento
 - ▶ Puede devolver funciones como resultado
- ▶ FD4 tiene funciones anidadas

FUNCIONES EN FD4

- ▶ FD4 tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumento
 - ▶ Puede devolver funciones como resultado
- ▶ FD4 tiene funciones anidadas
- ▶ ¿Cómo compilar funciones de FD4 a código de primer orden?

COMPILANDO FUNCIONES EN C

COMPILANDO FUNCIONES EN C

- ▶ El lenguaje C tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumentos (un puntero a una función)
 - ▶ Puede devolver funciones como resultado (un puntero a una función)

COMPILANDO FUNCIONES EN C

- ▶ El lenguaje C tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumentos (un puntero a una función)
 - ▶ Puede devolver funciones como resultado (un puntero a una función)
- ▶ Pero C no tiene funciones anidadas
 - ▶ Todas las funciones son top-level

COMPILANDO FUNCIONES EN C

- ▶ El lenguaje C tiene funciones de alto orden:
 - ▶ Puede recibir funciones como argumentos (un puntero a una función)
 - ▶ Puede devolver funciones como resultado (un puntero a una función)
- ▶ Pero C no tiene funciones anidadas
 - ▶ Todas las funciones son top-level
- ▶ Se compilan simplemente como punteros.

COMPILANDO FUNCIONES EN FD4

- Consideremos los programas

```
let suma : Nat -> Nat -> Nat =  
    fun (x : Nat) -> fun (y : Nat) -> x + y
```

```
let incrementar = suma 1
```

```
let decrementar = suma (-1)
```

- ¿Cómo compilar estas funciones?

REPRESENTACIÓN DE FUNCIONES EN C

REPRESENTACIÓN DE FUNCIONES EN C

- ▶ Cada función se representa como un puntero a un cacho de código que:
 - ▶ Espera su argumento en un registro `arg`;
 - ▶ computa el cuerpo de la función;
 - ▶ deja el resultado en un registro `res`;
 - ▶ vuelve al código llamador.

REPRESENTACIÓN DE FUNCIONES EN C

- ▶ Cada función se representa como un puntero a un cacho de código que:
 - ▶ Espera su argumento en un registro `arg`;
 - ▶ computa el cuerpo de la función;
 - ▶ deja el resultado en un registro `res`;
 - ▶ vuelve al código llamador.
- ▶ Si aplicamos este modelo, una función que devuelve una función debe generar dinámicamente código compilado

OPCIÓN 1: GENERACIÓN DE CÓDIGO

- ▶ suma genera código en runtime para cada argumento x

```
 $\mathcal{C}(\text{increment}) \rightarrow$  mov res,arg  
                        add res,1  
                        return
```

```
 $\mathcal{C}(\text{decrement}) \rightarrow$  mov res,arg  
                        add res,-1  
                        return
```

- ▶ Problema: generar código en runtime es complejo y caro

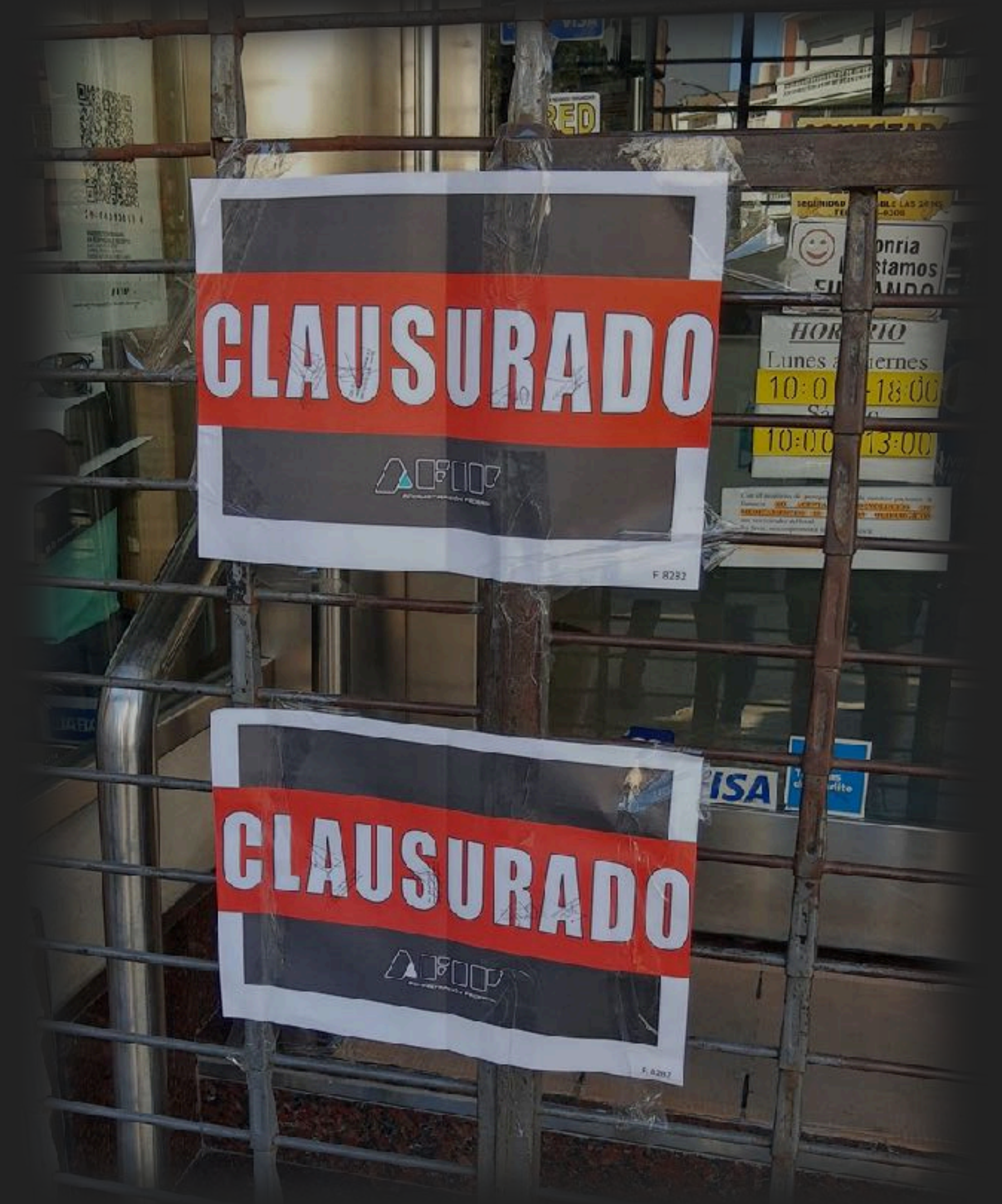
OPCIÓN 2: CLAUSURAS

- ▶ suma genera el código común:

```
mov res, arg  
add res, <valor del argumento x>  
return
```

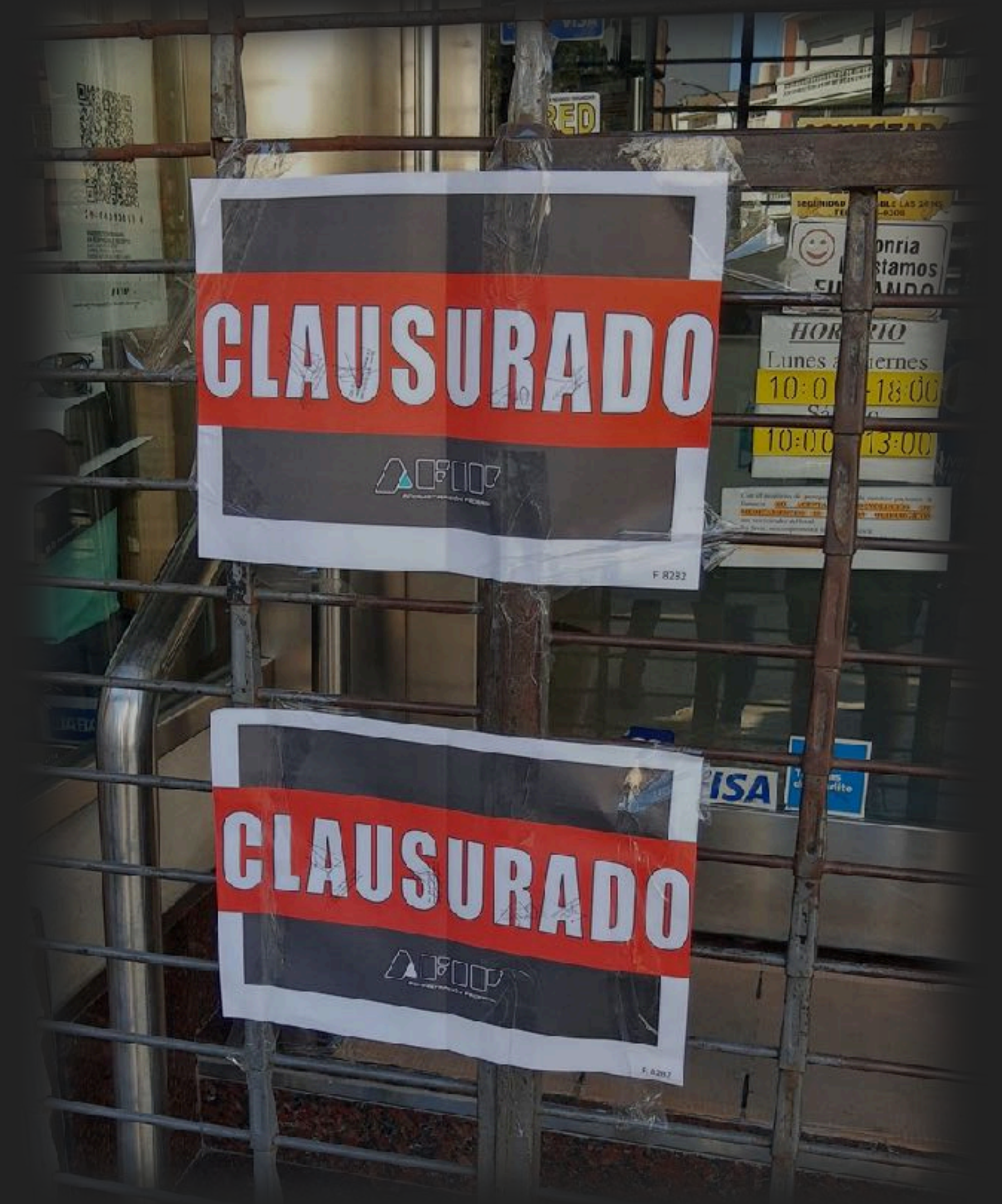
- ▶ Las partes que varían (las **variables libres**) se capturan en otra estructura:
el entorno

CLAUSURAS [LANDIN 1964]



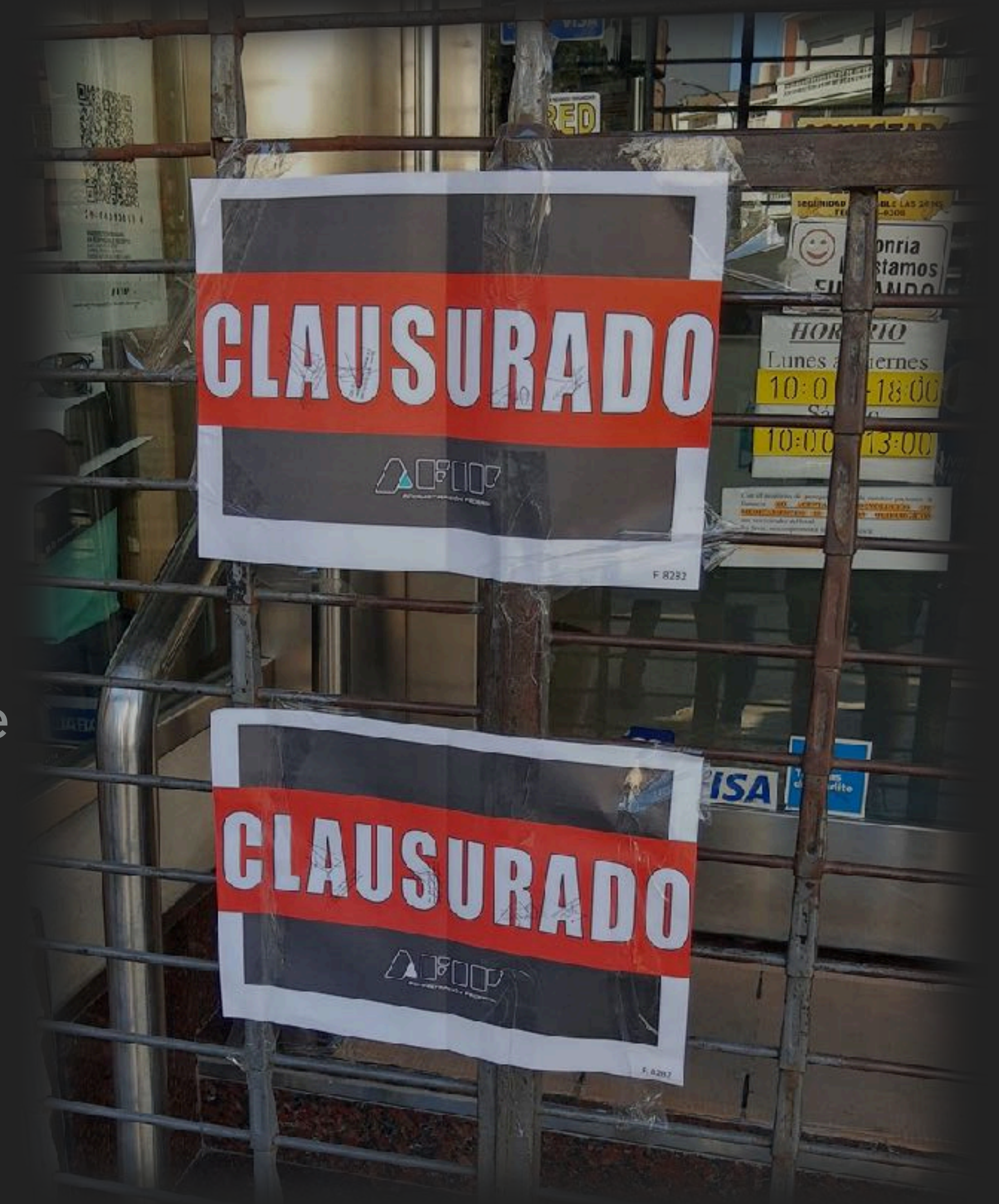
CLAUSURAS [LANDIN 1964]

- Todas las funciones quedan representadas por clausuras.



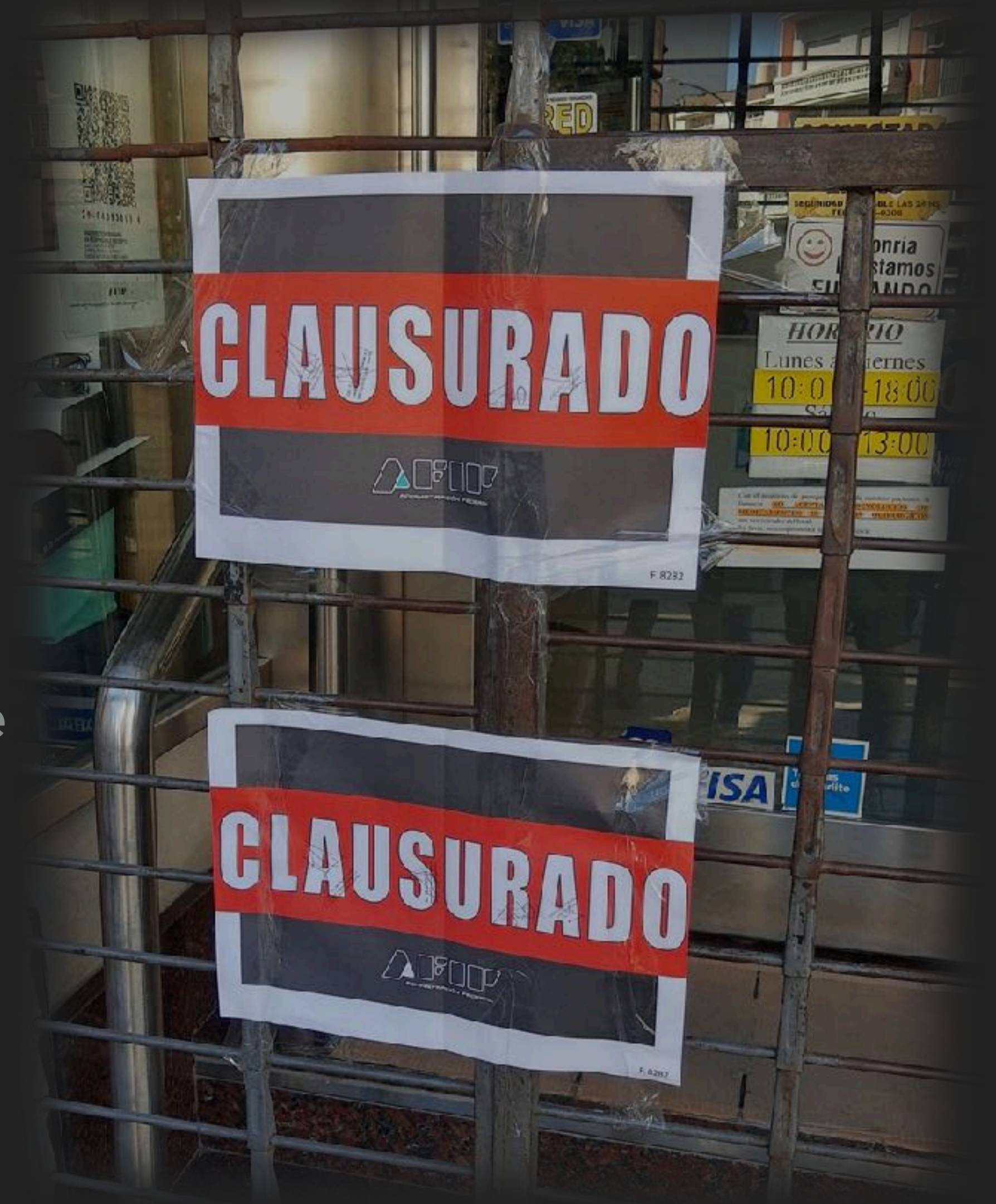
CLAUSURAS [LANDIN 1964]

- ▶ Todas las funciones quedan representadas por clausuras.
- ▶ Una clausura (*closure*) es una estructura en memoria dinámica que consiste de:
 - ▶ un puntero a código que computa el resultado
 - ▶ un entorno que le da significado a las variables libres de la función.



CLAUSURAS [LANDIN 1964]

- ▶ Todas las funciones quedan representadas por clausuras.
- ▶ Una clausura (*closure*) es una estructura en memoria dinámica que consiste de:
 - ▶ un puntero a código que computa el resultado
 - ▶ un entorno que le da significado a las variables libres de la función.
- ▶ Para aplicar la clausura, ponemos un puntero al entorno en el registro `env`, y llamamos al puntero.



increment

decrement



CÓDIGO DE EJEMPLO

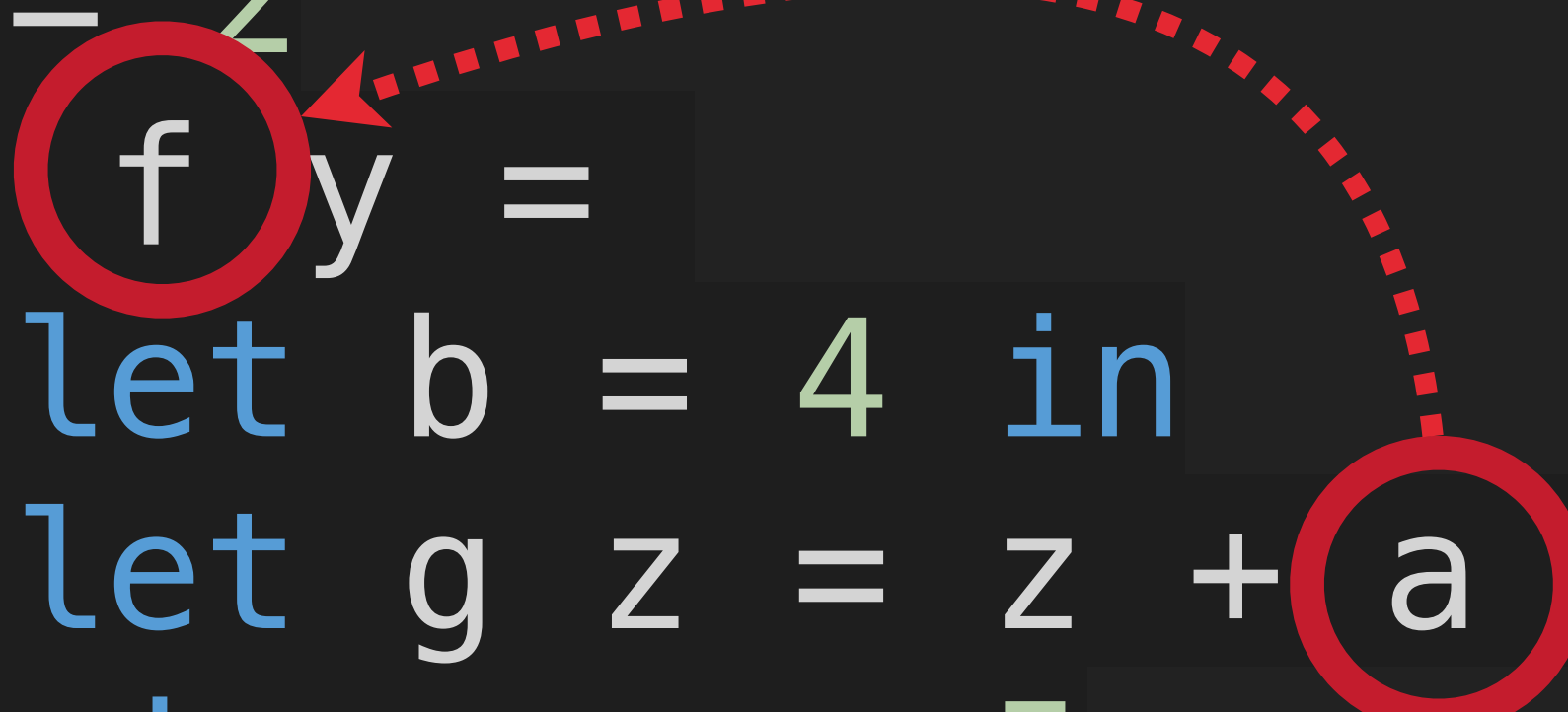
```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

CÓDIGO DE EJEMPLO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

CÓDIGO DE EJEMPLO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```



CÓDIGO DE EJEMPLO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```


CÓDIGO DE EJEMPLO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

CÓDIGO DE EJEMPLO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

The diagram illustrates the scope resolution for the variable `g` in the provided code. The variable `g` is circled in red. Two red dotted arrows originate from the circled `g` and point to the circled `a` and `b` in the expression `z + a + b`, indicating that `g` refers to the function defined in the innermost scope where `a` and `b` are defined.

EJEMPLO CONVERTIDO (APROXIMACIÓN CONCEPTUAL)

```
let {g [a, b]} z = z + a +  
b
```

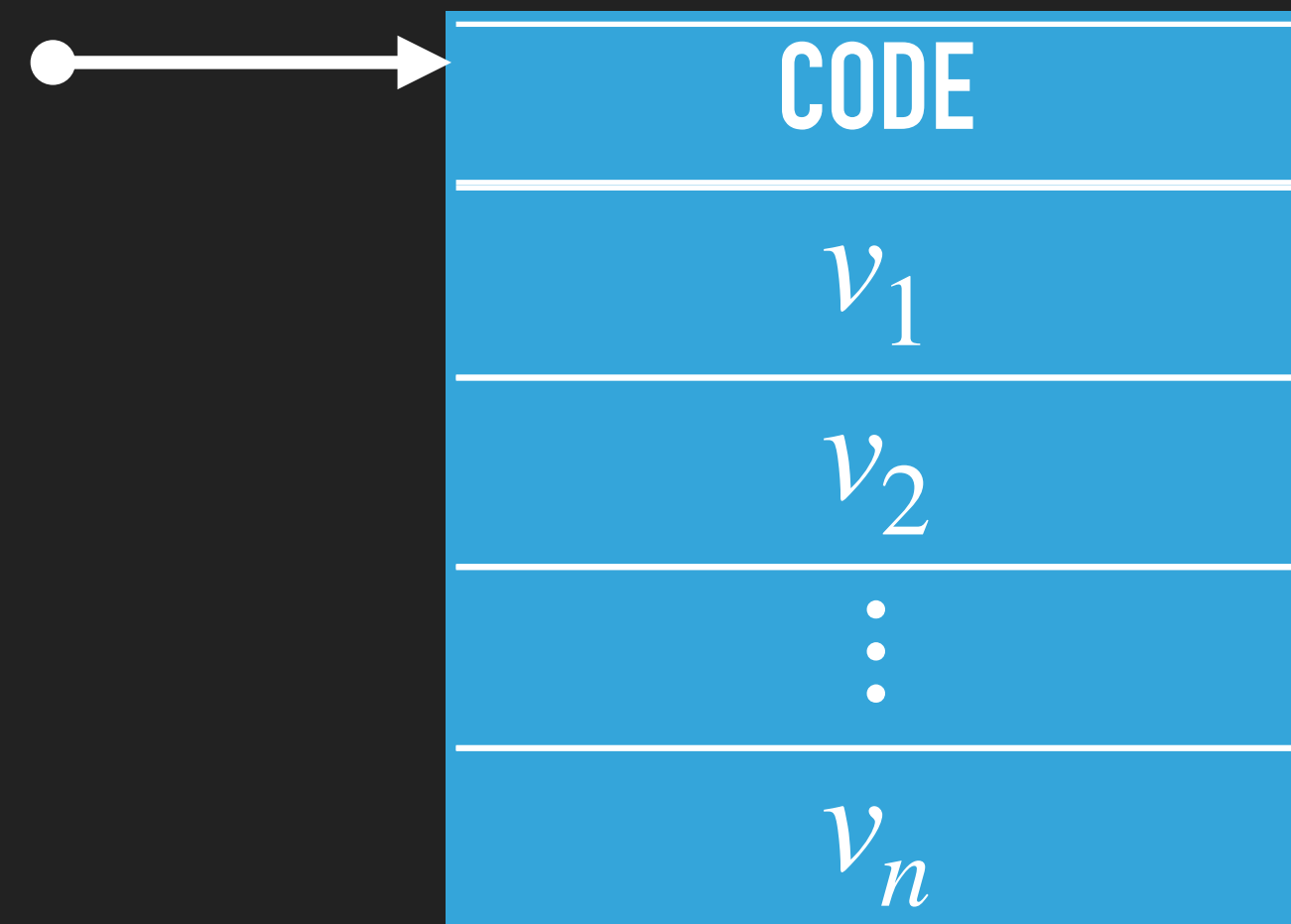
```
let {f [a]} y =  
    let b = 4 in  
    in y + {g [a, b]} 5
```

```
let a = 2  
in {f [a]} 3
```

REPRESENTACIÓN DE CLAUSURAS

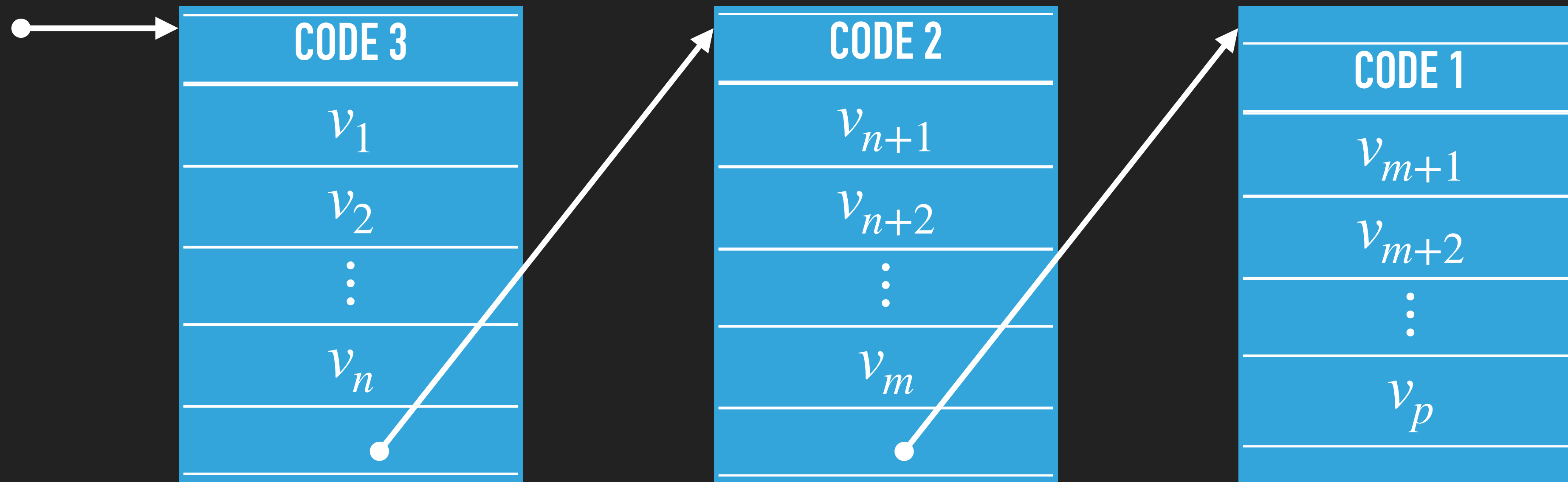
- ▶ En una aplicación, no se sabe nada acerca de la clausura que se llama (puede ser cualquier clausura del programa)
- ▶ El puntero a la función debe estar en una dirección conocida, para poder ser extraído.
- ▶ El contenido del entorno, sin embargo, no es usado en la aplicación. Sólo es usado en el cuerpo de la función. Esto nos da cierta libertad para representarlo.

CLAUSURAS PLANAS



El entorno está directamente en la clausura,
en vez de ser apuntado por la clausura

CLAUSURAS ENLAZADAS



Se reutilizan clausuras de las funciones de anidamiento superior

EJEMPLO CLAUSURA PLANA

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

g



CODE
$b \mapsto 4$
$a \mapsto 2$

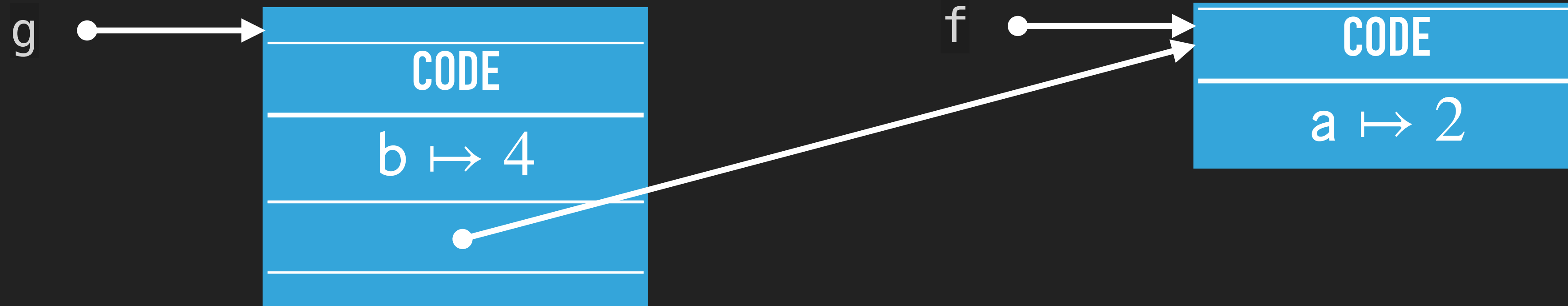
f



CODE
$a \mapsto 2$

EJEMPLO CLAUSURAS ENLAZADAS

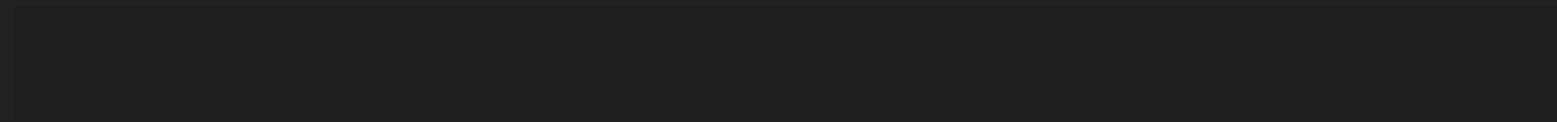
```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```



FUNCIONES RECURSIVAS

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura



FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```


FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

- ▶ El cuerpo de `f` tiene que pasar la clausura de `f` a la función `map`

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

- ▶ El cuerpo de `f` tiene que pasar la clausura de `f` a la función `map`
- ▶ Se puede lograr de diversas maneras:

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

- ▶ El cuerpo de `f` tiene que pasar la clausura de `f` a la función `map`
- ▶ Se puede lograr de diversas maneras:
 - ▶ Reconstruir la clausura de `f` a partir del entorno actual

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

- ▶ El cuerpo de f tiene que pasar la clausura de f a la función `map`
- ▶ Se puede lograr de diversas maneras:
 - ▶ Reconstruir la clausura de f a partir del entorno actual
 - ▶ Tratar a f como variables libre. En el entorno aparecerá un puntero a la clausura de f (clausura cíclica)

FUNCIONES RECURSIVAS

- ▶ Las funciones recursivas deben acceder a su propia clausura

```
let rec f = ... map f xs ...
```

- ▶ El cuerpo de f tiene que pasar la clausura de f a la función `map`
- ▶ Se puede lograr de diversas maneras:
 - ▶ Reconstruir la clausura de f a partir del entorno actual
 - ▶ Tratar a f como variables libre. En el entorno aparecerá un puntero a la clausura de f (clausura cíclica)
 - ▶ Usando clausuras planas, el entorno pasado a f es su propia clausura.

FUNCIONES MUTUAMENTE RECURSIVAS

- ▶ Las funciones mutuamente recursivas deben poder acceder a las clausuras de todas las funciones en la definición mutuamente recursiva.
- ▶ Supongamos que f y g son mutuamente recursivas. Las clausuras se pueden construir de dos maneras:
 - ▶ La clausura de f contiene un puntero a la clausura de g y viceversa (clausuras cíclicas)
 - ▶ Las funciones f y g comparten un clausura.

CONVERSIÓN DE CLAUSURAS

CONVERSIÓN DE CLAUSURAS

- ▶ Para compilar FD4 usaremos una transformación que se llama **conversión de clausuras** (*closure conversion*):

CONVERSIÓN DE CLAUSURAS

- ▶ Para compilar FD4 usaremos una transformación que se llama **conversión de clausuras** (*closure conversion*):
 - ▶ Se reemplazan las funciones por clausuras

CONVERSIÓN DE CLAUSURAS

- ▶ Para compilar FD4 usaremos una transformación que se llama **conversión de clausuras** (*closure conversion*):
 - ▶ Se reemplazan las funciones por clausuras
 - ▶ Se hace explícita la construcción, el pasaje y el acceso a entornos.

CONVERSIÓN DE CLAUSURAS

- ▶ Para compilar FD4 usaremos una transformación que se llama **conversión de clausuras** (*closure conversion*):
 - ▶ Se reemplazan las funciones por clausuras
 - ▶ Se hace explícita la construcción, el pasaje y el acceso a entornos.
- ▶ La transformación nos da como resultado código en que las funciones (al ser cerradas) se pueden representar por un puntero a código (como en C).

ALGORITMO DE CONVERSIÓN DE CLAUSURAS

► Variables

$$\llbracket x \rrbracket = x$$

► Constantes

$$\llbracket c \rrbracket = c$$

► Operadores

$$\llbracket op(x_1, \dots, x_n) \rrbracket = op(\llbracket x_1 \rrbracket, \dots, \llbracket x_n \rrbracket)$$

► Aplicaciones

$$\llbracket f x \rrbracket = \text{let } clos = \llbracket f \rrbracket \text{ in } clos[0] (clos, \llbracket x \rrbracket)$$

ALGORITMO DE CONVERSIÓN DE CLAUSURAS (CONT.)

► Funciones

$\llbracket \text{fun } x \rightarrow t \rrbracket = \text{makeBlock}(\text{codef}, v_1, \dots, v_k)$

donde:

- v_1, \dots, v_k son las variables libres de $\text{fun } x \rightarrow t$
- $\text{codef}(clos, x) = \text{let } v_1 = clos[1], \dots, v_k = clos[k] \text{ in } \llbracket t \rrbracket$
- El código codef es una función cerrada y por lo tanto puede ser "izada" a top-level.

EJEMPLO DE CÓDIGO CONVERTIDO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```


EJEMPLO DE CÓDIGO CONVERTIDO

```
let a = 2
in let f y =
    let b = 4 in
    let g z = z + a + b
    in y + g 5
in f 3
```

CLOSURE CONVERSION

```
let a = 2 in let f = <ff, a> in f[0] (f, 3)

let ff (clo, y) = let a = clo[1]
                  b = 4
                  g = <gg, a, b>
                  in y + g[0] (g, 5)

let gg (clo, z) = let a = clo[1]
                  b = clo[2]
                  in z + a + b
```

RESUMEN

- ▶ En C, todas las funciones son top-level y pueden ser representadas por punteros.
- ▶ En los lenguajes funcionales, como FD4, las funciones pueden ser anidadas y escapar el alcance del código que la crea.
- ▶ Por lo tanto las representamos con **clausuras**: un par de **un puntero a código y un entorno** que le da significado a sus variables libres
- ▶ Transformamos el código para usar explícitamente clausuras usando **conversión de clausuras**.