Reglas de Reducción y Programación Imperativa

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Razonamiento acerca de procedimientos

- Se introducen algunas reglas para razonar acerca de los programas
- Estas reglas permiten transformar procedimientos a formas cuya evaluación es más clara y eficiente

Razonamiento acerca de procedimientos

- Regla acerca de los procedimientos:
  - El resultado de una llamada a un procedimiento puede obtenerse substituyendo los operadores de la llamada por (en vez de) las variables en el cuerpo del procedimiento

```
> (define foo (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
    ⇒ ((lambda (x y) (+ (* x 3) y)) (+ 4 1) 7)
    ⇒ (+ (* (+ 4 1) 3) 7)
    ⇒ 22
```


Razonamiento acerca de procedimientos

- Regla alternativa:
  - El resultado de una llamada a un procedimiento puede obtenerse evaluando los argumentos y substituyendo sus valores por (en vez de) las variables en el cuerpo del procedimiento

```
> (define foo (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
    ⇒ (foo 5 7)          ;rep.literal del valor de la llamada
    ⇒ (+ (* 5 3) 7)
    ⇒ (+ 15 7)           ;rep.literal del valor de la llamada
    ⇒ 22
```

Razonamiento acerca de procedimientos

- Otro ejemplo:

```
> (define c+
    (lambda (n)
          (lambda (m) (+ n m))))
> ((c+ 5) 3)
    ⇒ ((lambda (m) (+ 5 m)) 3)
    ⇒ (+ 5 3)
    ⇒ 8
```


Evaluación compleja

• Regla de transformación de programas

(let ((var_1 exp_1) ... (var_n exp_n))
body)
⇒
(lambda (var_1,..., var_n) *body*) exp_1,..., exp_n)

• Evaluación de expresiones complejas:
- Reglas de transformación
- Substitución en llamadas a procedimientos

Razonamiento acerca de procedimientos

- Otra vez otro ejemplo:

```
> (let ((x 3)
        (add5 (c+ 5)))
    (add5 x))
```

- Substitución

```
⇒ (let ((x 3)
        (add5 ((lambda (m) (+ 5 m))))
    (add5 x))
```

- Transformación

```
⇒ ((lambda (x add5) (add5 x))      ; (var1,..., varn) body
   3                               ; exp1,..., expn
   (lambda (m) (+ 5 m)))
```


Razonamiento acerca de procedimientos

- Evaluación

```
⇒ ((lambda (x add5) (add5 x))
   3
   (lambda (m) (+ 5 m)))

⇒ ((lambda (m) (+ 5 m)) 3)
⇒ (+ 5 3)
⇒ 8
```


Razonamiento acerca de procedimientos

- Cuando substituimos se usa una rep. literal del valor de las exp.
- otro ejemplo:

```
> (let ((second (lambda (x) (car (cdr x))))
    (second (list 1 2 3)))
```

- Transformación

```
⇒ ((lambda (second) (second (list 1 2 3)))
          (lambda (x) (car (cdr x))))
⇒ ((lambda (x) (car (cdr x))) (list 1 2 3))
⇒ (car (cdr '(1 2 3))          ; se substituye por la literal y no por el valor
⇒ (car '(2 3))
⇒ 2
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Cadenas

El Cálculo Lambda y la conversión-β

- Sintaxis:
  - <exp> ::= <varref>
    | (lambda (<var>) <exp>)
    | (<exp> <exp>)
    | <number>
- Regla de transformación principal (evaluación):

  ((lambda (var) exp) rand) ⇒ exp[rand/var]

  - Expresión que resulta de substituir las ocurrencias de las variables *var* en *exp* por el argumento *rand*

El Cálculo Lambda y la conversión-β

- Conflictos de nombres de variables:

  > ((lambda (x)
  (lambda (y) (x y)))
  (y w))
  ⇒ (lambda (y) ((y w) y)) ?

  - La variable y en el argumento queda capturada por la parámetro formal del lambda anidado.
  - La referencia *y* en (*y w*) debe quedar libre después de la reducción
  - Solución: cambiar el nombre a la variable anidada (*y*), por otro nombre que no ocurra libre en el argumento (i.e., *z*).

El Cálculo Lambda y la conversión-β

- Cambio de nombre por la regla de conversión-α:

  > ((lambda (x)
  (lambda (z) (x z)))
  (y w))
  ⇒ (lambda (z) (y w) z))

  - Las variables *y* y *w* quedan libres en la expresión resultante.
- Conversión válida: substitución en la cual ninguna variable es capturada en el proceso de reducción

El Cálculo Lambda y la conversión-β

- Definición inductiva de la regla de substitución de *M* por *x* en *E*, *E*[*M*/*x*] (i.e., *M* en vez de *x*)
- *M* es el argumento de la función $\lambda x.E$
- La inducción se basa en la forma de *E*
  - <exp> ::= <varref>
    | (lambda (<var>) <exp>)
    | (<exp> <exp>)
    | <number>
- En notación del Cálculo Lambda
  - <exp> ::= <varref>
    | (λ <var>. <exp>)
    | (<exp> <exp>)
    | <number>

El Cálculo Lambda y la conversión-β

- Si *E* es la variable *x* ⇒ *x*[*M*/*x*] = *M*
  x[*y*/*x*] = y (i.e., *M* = *y*)
  x[(*lambda* (*x*) *x*)/*x*] = (lambda (x) x)
- Si *E* es otra variable *y* ⇒ *y*[*M*/*x*] = *y*
  y[*y*/*x*] = y
  y[*x*/*x*] = y

  Si *E* o una constante *c* ⇒ *c*[*M*/*x*] = *c*
  c[*y*/*x*] = c

El Cálculo Lambda y la conversión-β

- Si *E* es una aplicación funcional de forma (*F G*)
  ⇒ (*F G*)[*M*/*x*] = (*F*[*M*/*x*] *G*[*M*/*x*])
- Ejemplos:
  (x x) [*u*/*x*] = (u u)
  (x y) [*u*/*x*] = (u y)
  (x u) [*u*/*x*] = (u u)

El Cálculo Lambda y la conversión-β

- Si *E* es de la forma (lambda (*y*) *E*'):
  - Substituir para evitar que alguna variable sea capturada
  - Reducir: substituir el argumento por las referencias a la variable de referencia en el cuerpo de la función
- Notación:
  - Substitución: ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]
  - Si *E* es una variable, constante o aplicación funcional

    E[*M*/*x*] es como ya se definio!
  - Si *E* = (lambda (*x*) *E*') necesitamos definir:

    (lambda (*x*) *E*')[*M*/*x*]

El Cálculo Lambda y la conversión-β

Casos para la substitución:

(lambda (*y*) *E*')[*M*/*x*]

- *y* = *x* $\Rightarrow$ *x* no ocurre libre en *E*'
- *y* ≠ *x*
  - Si *y* ocurre libre en *M* & *x* ocurre libre en *E*' $\Rightarrow$ captura!
  - Si *y* no ocurre libre en *M* & *x* ocurre libre en *E*' $\Rightarrow$ *M* puede substituirse en *E*' de manera válida

El Cálculo Lambda y la conversión-β

- Si *E* es de la forma (lambda (*y*) *E*') & *y* = *x*

  (lambda (x) *E*')[*M*/*x*] = (lambda (x) *E*')
  - *y* es el parámetro formal de *E*
  - *x* es la variable a substituir en *E*'
- Ejemplos:

  ((lambda (x) (lambda (x) (+ x y))) 3) ; β-redex

  = (lambda (x) (+ x y))[3/x]

  = (lambda (x) (+ x y))

El Cálculo Lambda y la conversión-β

- Si *E* es de la forma (lambda (*y*) *E*') & *y* = x & *x* no ocurre libre en *E*':

  (lambda (x) *E*')[*M*/*x*] = (lambda (x) *E*')
  - *y* es el parámetro formal de *E*
  - *x* es la variable a substituir en *E*'
  - *x* no puede ocurrir libre en *E*
- Ejemplo:

  ((lambda (x) (lambda (x) x)) 3) ; β-redex

  = (lambda (x) x)[3/x]

  = (lambda (x) x)

El Cálculo Lambda y la conversión-β

- Si *E* es de la forma (lambda (*y*) *E*') & *y* ≠ *x* & *y* no ocurre libre en *M* (condición de reducción!):

  (lambda (*y*) *E*')[*M*/*x*] = (lambda (*y*) *E*'[*M*/*x*])
  - por lo tanto, *x* puede ocurrir libre en *E*' *y* podemos substituir *M* por *x* en *E*'
- Ejemplo:
  - ((lambda (x) (lambda (y) (+ x y))) 3) ; β-redex

    = (lambda (y) (+ x y))[3/*x*]

    = (lambda (y) (+ x y)[3/*x*])

    = (lambda (y) (+ 3 y))

El Cálculo Lambda y la conversión-β

- Si *E* es de la forma (lambda (*y*) *E*') & *y* ≠ *x* & *x* ocurre libre en *E*' & *y* ocurre libre en *M* (condición de captura!)

  (lambda (*y*) *E*')[*M*/*x*] = (lambda (*z*) *E*'[*z*/*y*])[*M*/*x*]

  donde *z* no ocurre libre en *E*' o en *M*
  - Ejemplo: ((lambda (x) (lambda (y) (x y))) (y w)) ; β-redex

    $\Rightarrow$ (lambda (y) (x y))[(y w)/x] ; condición de captura

    $\Rightarrow$ (lambda (z) (x y)[z/y])[(y w)/x]

    $\Rightarrow$ (lambda (z) (x z))[(y w)/x] ; z ≠ x & z no *OL* en M

    $\Rightarrow$ (lambda (z) (x z)[(y w)/x])

    $\Rightarrow$ (lambda (z) ((y w) z))

El Cálculo Lambda y la conversión-β

- Si se escogiera una variable que ocurre libre en *E'* (i.e., u):
  ((lambda (x) (lambda (y) (x y u))) (y w))
  ⇒ ((lambda (x) (lambda (u) (x u u))) (y w))
  Se capturaría *u* indebidamente!
- Si se escogiera una variable que ocurre libre en *M* (i.e., w):
  ((lambda (x) (lambda (y) (x y u))) (y w))
  ⇒ ((lambda (x) (lambda (w) (x w u))) (y w))
  ⇒ (lambda (w) ((y w) w u))
  Se capturaría *w* indebidamente!

El Cálculo Lambda y la conversión-β

- β-redex: expresión de forma ((lambda (x) *E*) *M*)
- conversión-β:
  ((lambda (x) *E*) *M*) = *E*[*M*/*x*]
- reducción-β:
  - Cuando la conversión-β se utiliza o aplica de izquierda a derecha (función a la izquierda y argumento a la derecha) para transformar un β-redex
  - La expresión se reduce a su forma más "simple".

El Cálculo Lambda y la conversión-β

- Reducir:
  ((lambda (x) (x (y x))) z)
- Forma normal redex: ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]
  ⇒ (x (y x))[z/x] ; (*F G*)[*M*/*x*] = (*F*[*M*/*x*] *G*[*M*/*x*])
  ⇒ (x[z/x] (y x)[z/x]) ; *x*[*M*/*x*] = *M* & (*F G*)[*M*/*x*]
  ⇒ (z (y[z/x] x[z/x])) ; *y*[*M*/*x*] = *y* & *x*[*M*/*x*] = *M*
  ⇒ (z (y z))

El Cálculo Lambda y la conversión-β

- Reducir:
  ((lambda (x) (x y)) (lambda (y) (x y)))
- Forma normal redex: ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]
  ⇒ (x y)[(lambda (y) (x y))/x] ; (*F G*)[*M*/*x*] = (*F*[*M*/*x*] *G*[*M*/*x*])
  ⇒ (x[(lambda (y) (x y))/x] y[(lambda (y) (x y))/x])
  ; *x*[*M*/*x*] = *M* & *y*[*M*/*x*] = *y*
  ⇒ ((lambda (y) (x y)) y) ;Forma normal redex
  ⇒ (x y)[y/y] ; (*F G*)[*M*/*x*] = (*F*[*M*/*x*] *G*[*M*/*x*])
  ⇒ (x[y/y] y[y/y]) ; *y*[*M*/*x*] = *y* & *x*[*M*/*x*] = *M*
  ⇒ (x y)

El Cálculo Lambda y la conversión-β

- Reducir:
  ((lambda (x) (lambda (y) ((x y) z))) (lambda (a) y))
- Forma normal redex: ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]
  ⇒ (lambda (y) ((x y) z))[(lambda (a) y)/x]
  Si *E* es de la forma (lambda (*y*) *E'*) & *y* ≠ *x* & *x* ocurre libre en *E'* & *y* ocurre libre en *M*:
  (lambda (*y*) *E'*)[*M*/*x*] = (lambda (*z*) *E'*[*z*/*y*])[*M*/*x*]
  ⇒ (lambda (u) ((x y) z))[u/y])[(lambda (a) y)/x]
  ⇒ (lambda (u) ((x y)[u/y] z[u/y])[(lambda (a) y)/x]

El Cálculo Lambda y la conversión-β

⇒ (lambda (u) ((x[u/y] y[u/y]) z))[(lambda (a) y)/x]

⇒ (lambda (u) ((x u) z))[(lambda (a) y)/x]

E es de la forma (lambda (*u*) *E'*) & *u* ≠ *x* & *u* no ocurre libre en *M*:

(lambda (*u*) *E'*)[*M*/*x*] = (lambda (*u*) *E'*[*M*/*x*])

⇒ (lambda (u) (((lambda (a) y) u) z)) ;forma normal redex

⇒ (lambda (u) (y[u/a] z))

⇒ (lambda (u) (y z))

El Cálculo Lambda y la conversión-β

- Reducir:

```
((lambda (x)
   (lambda (y)
      ((lambda (x) (z x))
       (lambda (y) (z y)))))
 (lambda (y) y))
```

- Forma normal redex: ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]

El Cálculo Lambda y la conversión-β

⇒ (lambda (y)

```
((lambda (x) (z x))
 (lambda (y) (z y)))) [(lambda (y) y)/x]
```

E es de la forma (lambda (*y*) *E*') & $y \neq x$ & *y* no ocurre libre en *M*:

(lambda (*y*) *E*')[*M*/*x*] = (lambda (*y*) *E*'[*M*/*x*])

⇒ (lambda (y)

```
((lambda (x) (z x))
 (lambda (y) (z y))) [(lambda (y) y)/x])
```


El Cálculo Lambda y la conversión-β

⇒ (lambda (y)

```
((lambda (x) (z x))[(lambda (y) y)/x]))
 (lambda (y) (z y))[(lambda (y) y)/x])
```

⇒ (lambda (y)

```
((lambda (x) (z x))
 (lambda (y) (z y)))
```

⇒ (lambda (y)

```
((z (lambda (y) (z y))))
```


El Cálculo Lambda y la conversión-β

- La **conversión-β** se puede también aplicar de derecha a izquierda (argumento a la izquierda y función a la derecha), como en el *let*:
  - ((f (a (b c))) (a (b c)))
    ⇒ ((lambda (x) ((f x) x)) (a (b c)))
    ⇒ (let ((x (a (b c)))) ((f x) x))

El Cálculo Lambda y la conversión-β

- La conversión-η (un solo argumento):

  ((lambda (x) (*E* x)) = *E*

  - *E* denota una función de una variable
  - *x* no ocurre libre en *E*
  - Lado izquierdo: η-redex

conversión-eta

- Consideremos el siguiente patrón:

```
(define extend-ff
        (lambda (sym val ff)
                (make-entended-ff sym val ff)
```

  Equivalente a:

```
(define extend-ff make-extended-ff)
```

- Otros ejemplos:

  (lambda (x) (sin x)) = sin

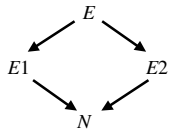
  (lambda (x) ((lambda (y) y) x))) = (lambda (y) y)

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (*applicative-order reduction*)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Cadenas

Estrategias de reducción

- Problema: Una expresión puede tener más de un *redex* y puede haber más de uno modo de reducirla o evaluarla:
  - Substituir o evaluar los argumentos? (ejemplo inicial)
- Teorema de Church-Rosser (confluencia o propiedad del diamante):
  - Si E puede reducirse a $E1$ o $E2$ entonces existe una expresión N que puede obtenerse tanto de $E1$ como de $E2$



Estrategias de reducción

- Puede una expresión reducirse a más de una constante?
  - Por Church-Rosser tendría que haber un término N al cual estas constantes tendrían que ser reducidas
  - Las constantes no pueden ser reducidas
  - Por lo tanto, una expresión no puede reducirse a más de una constante!
- Garantiza Church-Rosser que mediante la aplicación de la reducción-β se obtiene una constante o una "expresión básica"?

Estrategias de reducción

- Garantiza Church-Rosser que la aplicación de la reducción-β obtiene una *respuesta*?
- Garantiza Church-Rosser que la aplicación de la reducción-β *para*?
- Qué es una respuesta?
  - Una constante?
- No toda expresión se puede reducir a una constante:

  ((lambda (x) (x x)) (lambda (x) (x x)))

  $\Rightarrow$ ((lambda (x) (x x)) (lambda (x) (x x)))

Estrategias de reducción

- Condición más debil: garantiza la aplicación de la reducción-β que se obtendrá una respuesta si esta existe?
  - Algunas expresiones se reducen a constantes, pero las secuencias de reducción pueden ser infinitas:

    ((lambda (y) 3)

    ((lambda (x) (x x)) (lambda (x) (x x))))
  - Reduciendo y $\Rightarrow$ 3 ; $c[M/x] = c$
  - Reduciendo x $\Rightarrow$ secuencia de reducción infinita
- Las estrategias de evaluación pueden diferir en cuanto a que pueden o no encontrar una respuesta, pero Church-Rosser garantiza que si se encuentra (si termina) la respuesta será siempre la misma.

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (*applicative-order reduction*)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Cadenas

Reducción por orden de aplicación

- Reducción en orden de aplicación (RPOA): Scheme
- Una *respuesta* (valor de una expressión) es:
  - Constante
  - Variable
  - Expresión lambda
  - NUNCA una aplicación: (*Exp Exp*)

Reducción por orden de aplicación

- La reducción-β (RPOA) sólo se aplica si:
  - El operando y operador son respuestas
  - En caso contrario el operando y el operador se reducen
  - Un redex en el cuerpo de una abstracción nunca se reduce
    (lambda (x) (lambda (y) y) 3))
  - No se aplica la reducción-η
    (lambda (x) ((lambda (y) y) x))) = (lambda (y) y)
- Estás convenciones se conocen como:
  Lambda-value calculus (Cálculo lambda-valor?)

Reducción por orden de aplicación

- β-*redex*: ((lambda (x) *E*) *M*)
- La reducción-β en RPOA (en Scheme)
  ((lambda (*x*) *E*) *M*) = *E*[*M*/*x*]
  donde *M* es una respuesta (valor)
- Ejemplo:
  - (lambda (x) (x (x y))) (lambda (w) w) z))
    E = (x (x y))
    M = (lambda (w) w) z)) ;no es un resultado
    Por lo tanto hay que reducir el operando
  - (lambda (x) (x (x y))) z)) ;op. es un resultado
  - (z (z y)))

Reducción por orden de aplicación

- Procedimiento para RPOA: reduce una aplicación (*reduce-once-appl*):
  - Toma como entrada una expresión parseada
  - Reduce el primer redex que encuentra
  - Regresa la expresión reducida (completa)
  - Si la expresión es una contante, una variable o una exp-lambda no se puede reducir y la computación termina
- El interprete:
  - Aplica *reduce-once-appl* hasta que todos los redex han sido reducidos

Reducción por orden de aplicación

- Definición de *reduce-once-appl*
  - Si es un β-*redex* simplemente se realiza la reducción
    - Si no se trata de reducir el operador
    - Si no se trata de reducir el operando
  - Cuando se reduce una sub-expresión hay que construir una nueva expresión con la reducción

Reducción por orden de aplicación

- *Continuators*: procedimientos que indican como continúa la computación
  - *succeed*: recibe como argumento la expresión que resulta de una reducción exitosa e indica que hacer
  - *fail*: no tiene argumentos y se invoca si la computación falla (si la expresión a ser reducida es un resultado)
- Especificación de *reduce-once-appl*:
  (reduce-once-appl exp succeed fail) =
  - (*succeed exp'*) si *exp* contiene un β-*redex* aplicativo, donde *exp'* es el resultado de reducir *exp* en forma aplicativa
  - (*fail)* si *exp* no tiene ningún β-*redex* aplicativo

Definición de *reduce-once-appl*

```
> (define answer?  (lambda (exp) (not app? exp))))
> (define reduce-once-appl
    (lambda (exp succeed fail)
      (variant-case exp
          (lit (datum) (fail))
          (varref (var) (fail))
          (lambda (formal body) (fail))
          (app (rator rand)
              (if (and (beta-redex? exp) (answer? rand))
                  (succeed (beta-reduce exp))
                  ...
```



```
... (reduce-once-appl                    ;proc. fail: reduce-rator)
        rator
        (lambda (reduced-rator)          ; succeed-rator
            (succeed
                (make-app reduced-rator rand)))
        (lambda ( )                      ;fail-rator
            (reduce-once-appl            ;reduce-rand
                rand
                (lambda (reduced-rand)           ;succeed-rand
                    (succeed
                        (make-app rator reduced-rand)))
                fail))))))))                     ;fail-rand
```


Uso de *reduce-once-appl*

```
> (define succeed
    (lambda (exp)
      (variant-case exp
          (lit (datum) (make-lit datum))
          (varref (var) (make-varref var))
          (lambda (formal body) (make-lambda formal body))
          (else exp)))) ;regresa el registro tipo aplicación

> (define fail (lambda ( ) 'fail))
```


Uso de *reduce-once-appl*

```
> (reduce-once-appl 3 succeed fail)
> fail
> (reduce-once-appl x succeed fail)
> fail
> (reduce-once-appl (lambda (x) x) succeed fail)
> fail
> (reduce-once-appl ((lambda (x) x) 3) succeed fail)
> #(lit 3)     ;representación del registro como vector
```


Uso de *reduce-once-appl*

```
> (reduce-once-appl ((lambda (x) x) ((lambda (y) y) 3))) succeed fail)
⇒   (lambda ((lambda (x) x) ((lambda (y) y) 3))) succeed fail)
      (variant-case exp
          ...
          (app ((lambda (x) x) ((lambda (y) y) 3)))
              (if (and (beta-redex? (lambda (x) x))) (answer? rand))
                      ... No: rand no es una respuetsa
                  (succeed (beta-reduce exp))
                  ...
```



```
... (reduce-once-appl                    ;reduce rator!
        (lambda (x) x)
        (lambda (reduced-rator)          ;succeed embebido
            (succeed                     ;succeed inicial!
                (make-app reduced-rator rand)))
        (lambda ( )                      ;fail embebido (reduce rand)
            (reduce-once-appl
                ((lambda (y) y) 3))
                (lambda (reduced-rand)
                    (succeed
                        (make-app (lambda (x) x) reduced-rand)))
                fail))))))))
```

Definición de *reduce-once-appl*

```
> (define reduce-once-appl
    (lambda ((lambda (x) x) succeed-rator fail-rator)
      (variant-case (lambda (x) x)
            ...
            (lambda ((x) x) (fail-rator))
            ...
```

Pero quien es el nuevo *fail*?


```
... (reduce-once-appl
        (lambda (x) x)
        (lambda (reduced-rator)        ;succeed embebido
            (succeed                   ;succeed inicial!
                (make-app reduced-rator rand)))
        (lambda ( )                    ;fail-rator
            (reduce-once-appl
                ((lambda (y) y) 3))
                (lambda (reduced-rand)
                    (succeed
                        (make-app (lambda (x) x) reduced-rand)))
                fail))))))))
```



```
⇒ (reduce-once-appl                ;...la nueva llamada!
        ((lambda (y) y) 3))
        (lambda (reduced-rand)         ;succeed-rand
            (succeed
                (make-app (lambda (x) x) reduced-rand)))
        fail))))))))

⇒ (reduced-once-apply              ... la  nueva ejecución!
    (variant-case ((lambda (y) y) 3))
        ..
        (app (rator rand)
            (if (and (beta-redex? exp) (answer? rand))
                (succeed (beta-reduce exp))
```



```
⇒ (reduced-once-apply              ... la  nueva ejecución!
    (variant-case ((lambda (y) y) 3))
        ...
        (succeed (beta-reduce ((lambda (y) y) 3))))

⇒ (lambda (3)      ;succeed-rand
        (succeed     ;succeed original
            (make-app (lambda (x) x) 3)))

⇒ #(app (lambda (x) x) 3)
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Reducción por la izquierda

- Reducción de orden aplicativo no termina siempre:

```
((lambda (y) 3)
        ((lambda (x) (x x)) (lambda (x) (x x))))
```

  M = "((lambda (x) (x x)) (lambda (x) (x x)))" no es un valor!
- Reducción por la izquierda:
  - Reduce el β-redex cuyo paréntesis izquierdo aparece primero

```
((lambda (y) 3)
        ((lambda (x) (x x)) (lambda (x) (x x))))
```

  - garantiza terminar

Reducción por la izquierda

- Forma normal: no contiene ningún β-redex
  - No se puede reducir más:
    i.e., (lambda (x) x)
  - Por Church-Rosser: toda expresión tiene sólo una forma normal
  - Se puede probar que la reducción por la izquierda siempre encuentra la forma normal, si ésta existe.
- Reducción normal: Reducción por la izquierda

Reducción por la izquierda

- El costo es la eficiencia:
  (lambda (x) (x (x y))) (lambda (w) w) z))
  ⇒ ((lambda (w) w) z) (((lambda (w) w) z) y))
  ⇒ (z (((lambda (w) w) z) y))
  ⇒ (z (z y))
- Pero usando reducción de orden aplicativo:
  (lambda (x) (x (x y))) (lambda (w) w) z))
  ⇒ (lambda (x) (x (x y))) z)
  ⇒ (z (z y))

Reducción por la izquierda

- La mayoria de los lenguajes usan orden aplicativo
- Algunos lenguaje usan reducción izquierda para encontrar una forma normal, si es que existe.

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Cadenas

Procedimientos Recursivos

- En el cálculo lambda "puro" no hay *define* o *letrect*
- Implementación de la recursión:
  - exp: expresión lambda que define un procedimiento recursivo
  - *g*: variable libre en exp, con referencia a la cual se hace la recursión (*g* es un nombre para *exp*)
  - forma de la recursión:
    (lambda (g) *exp*)
- Extendiendo el cálculo lambda con:
  - if, zero?, *, -, 1 y g (ligada a la función factorial)

Procedimientos Recursivos

- Definición del factorial:

```
(lambda (g)
        (lambda (n)
                (if (zero? n) 1 (* n (g  (- n 1))))))
```

- Si pasamos la función *g* como parámetro se obtiene una función que recibe un argumento *n* y que aplica a *g* dentro de su cuerpo
- La forma de la recursión es:
  ((g g) n) = (f n)
- Pero *f* contiene a *g*!
- *g* tiene que volverse a aplicar a menos que *n* sea 1!

Procedimientos Recursivos

- La secuencia de aplicación se repite n - 1 veces!

  (f n) ⇒ ((g g) n)

  (f (f n)) ⇒ ((g g) ((g g) n))

  (f (f (f n))) ⇒ ((g g) ((g g) ((g g) n)))

  ...

  (f (f ... (f n))) ⇒((g g) ((g g) ... ((g g) n)))

- Para definir la recursión se necesita un operador que reproduzca la función en cada aplicación

  ⇒(f ((g g) ... (f n)))

  ⇒(f (Y (f n)))

Procedimientos Recursivos

- Combinador Y:

  (Y f)

- La recursión se obtiene ligando g a (Y f) aplicando:

  (f (Y f))

- Como f regresa el procedimiento recursivo

  (Y f) = (f (Y f))

- Definición del combinador Y (evaluación izquierda):

```
(lambda (f)
    ((lambda (x) (f  (x x)))
     (lambda (x) (f (x x)))))
```


Procedimientos Recursivos

- Aplicación (Y f) en orden izquierdo:

```
((lambda (f)
    ((lambda (x) (f  (x x)))
     (lambda (x) (f (x x)))))
        (lambda (g)
            (lambda (n)
                (if (zero? n) 1 (* n (g  (- n 1)))))))
```


Procedimientos Recursivos

- Aplicación (Y f) en orden izquierdo:

```
((lambda (f)
    ((lambda (x) (f  (x x)))
     (lambda (x) (f (x x)))))
            fact)
```


Procedimientos Recursivos

- Aplicación (Y f) en orden izquierdo:

```
((lambda (x) (fact (x x)))
 (lambda (x) (fact (x x))))
```


Procedimientos Recursivos

- Aplicación (Y f) en orden izquierdo:

```
((lambda (x) (fact (x x)))
 (lambda (x) (fact (x x))))
```

Procedimientos Recursivos

- Aplicación (Y f) en orden izquierdo:

```
((lambda (x) (fact (x x)))
 (lambda (x) (fact (x x))))
⇒
(fact ((lambda (x) (fact (x x)))
       (lambda (x) (fact (x x)))))
```


Procedimientos Recursivos

- Evaluación (orden izquierdo):
  ((Y f) n)
- para $n = 2$
  ((fact ((lambda (x) (fact (x x)))
  (lambda (x) (fact (x x))))) 2)
- "(fact (x x))" regresa
  (lambda (n)
  (if (zero? n) 1 (* n (g (- n 1)))))
- Esquemáticamente:
  (f (f n)) ⇒ ((g g) ((g g) 2))

Procedimientos Recursivos

- Aplicación (Y f) en orden aplicativo usando Scheme:

```
> (define f
    (lambda (g)
      (lambda (n)
        (if (zero? n) 1 (* n (g (- n 1)))))))
> (define Y
    ((lambda (f)
       ((lambda (x) (f (lambda (y) ((x x) y))))
        (lambda (x) (f (lambda (y) ((x x) y)))))))
```


Procedimientos Recursivos

- Aplicación (Y f) en orden aplicativo usando Scheme:

```
> (define factorial
    (lambda (n) ((Y f) n)))

> (factorial 2)
2
> (factorial 3)
6
```


Procedimientos Recursivos

- Aplicación ((Y f) 2) :

```
(((lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
      (lambda (g)
        (lambda (n)
          (if (zero? n) 1 (* n (g (- n 1))))))) 2)
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Programación Funcional (Scheme... hasta ahora):

- Ventajas:
  - No hay efectos laterales a las evaluaciones
  - Fácil de visualizar procedimientos de primer orden
  - Fácil de razonar acerca de los procedimientos
  - Programación muy elegante
- Desventaja:
  - "Elevada"
  - Ineficiente!

Programación Imperativa

- Características:
  - Algunas operaciones se ejecutan no por el valor de la función sino más bien por los efectos laterales
  - Asignación de valores a variables (o localidades)
  - Entrada y salida (Input/output)
  - Se requiere introducir mecanismos de secuenciación
- Desventajas:
  - Código complejo
  - Difícil razonar acerca del código (i.e., conversión-β)

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Secuenciación

- Orden operaciones con efectos laterales (statements)
- En Scheme:
  (begin exp_1 exp_2 ... exp_n) = exp_n
- Los valores de las otras expresiones se descartan
- Ejemplo:
  - *display* ⇒ imprime su argumento (sin comillas)
  - *newline* ⇒ escribe un new-line
- Los valores de procedimientos estándar cuyo solo propósito es producir efectos laterales no se especifican ya que no se requieren

Secuenciación (ejemplos)

- Efectos del valor de *begin*:

```
> (begin (display "dos mas dos = ") (display (+ 2 2)))
dos mas dos = 44
```

- el segundo "4": resultado del valor de *begin*
- corrigiendo

```
> (begin (display "dos mas dos = ")
  (display (+ 2 2))
  (newline))
dos mas dos = 4
>
```


for-each

- Similar a *map*, pero evalúa argumentos de izquierda a derecha

```
> (define for-each
    (lambda (proc lst)
      (if (null? lst)
          'done
          (begin
            (proc (car lst))
            (for-each proc (cdr lst))))))
```

for-each

- Ejemplo: desplegar argumentos en orden

```
> (define displayln
     (lambda lst
          (begin
               (for-each display lst)
               (newline))))
> (displayln 2 “+” 2 “=” (+ 2 2))
2+2=4
>
```


begin implícito

- *begin* puede ser implícito:

```
> (define lst
     (begin
          (for-each display lst)
          (newline)))
```

equivalente a:

```
> (define lst
     (for-each display lst)
     (newline)))
```


for-each implícito

- otro ejemplo:

```
> (case salta-veces
     ((1 uno) (newline))
     ((2 dos) (begin (newline) (newline)))
     (else (begin (newline) (newline) (newline))))
```

es equivalente a:

```
> (case salta-veces
     ((1 uno) (newline))
     ((2 dos) (newline) (newline))
     (else (newline) (newline) (newline)))
```


begin implícito y debugging

```
> (define fact
    (lambda (n)
      (displayln “entrando a fact con n = ” n)
      (let ((resp (if (zero? n)
                      1
                      (* n (fact (- n 1))))))
        (displayln “saliendo de fact con” resp)
        resp)))
```


for-each implícito

```
> (fact 3)
entrando a fact con n = 3
entrando a fact con n = 2
entrando a fact con n = 1
entrando a fact con n = 0
saliendo de fact con 1
saliendo de fact con 1
saliendo de fact con 2
saliendo de fact con 6
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción por orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

salida (output)

- *write*: imprime argumentos en su representación literal

```
> (begin
        (write "cadena escrita")
        (display " cadena escrita ")
        (write #\x)
        (display #\space)
        (display #\x)
        (display #\newline))
  "cadena escrita" cadena escrita #\x x
>
```


Entrada (input)

- Lectura
  - por carácter: read-char
  - por expresión: read
- No tienen argumentos y su valor es lo que se lee
- Si *read* o *read-char* encuentran un *eof* en lectura se regresa un objeto tal que:

  (eof-objetc? (read)) = #t

  para cualquier valor que puede ser leido:

  (eof-object? (read)) = #f
- Los procedimiento estandar de entrada y salida pueden tomar argumentos para redireccionar la entrada y/o salida
  - Defaults: teclado y pantalla

Ciclo de lectura-evaluación-escritura

```
> (define read-eval-print
    (lambda ( )
        (display "-->")
        (write (eval (read)))
        (newline)
        (read-eval-print)))
```

- Cualquier procedimiento con errores aborta, y el control recae en el ciclo *read-eval-print*

Ciclo de lectura-evaluación-escritura

```
> (read-eval-print)
--> (+ 1 2)
3
--> (car (cons "foo" 'foo))
"foo"
--> (cdr 3)
Error: Invalid argument to cdr: 3
>              ; se regresa al prompt estándar
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Cadenas

Mutación de estructuras de datos

- Los efectos laterales se reflejan en cambiar los datos mediante procedimientos destructivos
  - set-car!
  - set-cdr!
  - set-vector!
- Elementos "mutables": cons pairs, vectores, cadenas
- Elementos "inmutables": símbolos, números

Mutación de estructuras de datos

- Ejemplos

```
> (define lst (list 1 2))
> (set-car! lst 3)
> lst
(3 2)
> (set-cdr! lst (list 4 5))
> lst
(3 4 5)
> (set-cdr! lst 6)
> lst
(3 . 6)
```


Mutación de estructuras de datos

- vector-set!
  - recibe un vector, un indice (base cero) y un nuevo valor
- ejemplo:

```
> (define v (vector 1 2 3))
> (vector-set! v 1 4)
> v
#(1 4 3)
```


Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Asignación y compartición (sharing) de variables

- Programación funcional: variables ligadas a valores
- Programación imperativa: variables ligadas a localidades
- Asignación de variables: (set! *var exp*)
  - se evalúa *exp*
  - se asigna el valor a la localidad referida por *var*
  - las variables deben ser definidas (define) antes de ser modificadas (set)

Asignación y compartición (sharing) de variables

- Ejemplo:

```
> (define x 1)
> (set! x 2)
> x
2
> (let ((y 3))
        (set! y 4)
        (+ y 1))
5
```


Asignación y compartición (sharing) de variables

```
> (define a '*)
> (let ((b 'no-tres))
      (displayln "b es  " b)
      (set! a 3)
      (if (= a 3) (set! b 'tres))
       b)
b es no-tres
tres
> a
3
```

Definición de un stack... ligas compartidas

```
> (define empty? '*)
> (define push! '*)
> (define pop! '*)
> (define top '*)
> (let ((stk '( )))
      (set! empty? (lambda ( ) (null? stk)))
      (set! push! (lambda (x) (set! stk (cons x stk))))
      (set! pop! (lambda ( ) (if (empty?)
                                 (error "stack vacio")
                                 (set! stk (cdr stk)))))
      (set! top! (lambda ( ) (if  (empty?)
                                 (error "stack vacio")
                                  (car stk)))))
```


Definición de un stack... ligas compartidas

```
> (push! 1)
> (push! 2)
> (top)
2
> (pop!)
> (top)
1
> (pop!)
```

- Los procedimientos *empty?, push!*, *pop!* y *top* comparten la misma liga a *stk*
- Esta liga es local y el stack es una abstracción de datos: *stk* se encuentra encapsulada

Definición de un stack... con mensajes!

```
> (define stack
      (let ((stk '( )))
         (lambda (message)
              (case message
                   ((empty?) (lambda ( ) (null? stk)))
                   ((push!) (lambda (x) (set! stk (cons x stk))))
                   ((pop!) (lambda ( ) (if (null? stk)
                                           (error "stack vacio")
                                           (set! stk (cdr stk)))))
                   ((top!) (lambda ( ) (if  (null? stk)
                                           (error "stack vacio")
                                           (car stk))))
                   (else (error "stack: mensaje invalido" message))))))
```


Definición de un stack... ligas compartidas

- La procedimiento lambda se encuentra en el cuerpo del *let*
- Por lo tanto, los proc. del stack se refieren a la liga compartida

```
> ((stack 'push!) 1)
> ((stack 'push!) 2)
> ((stack 'top))
2
> ((stack 'pop!))
> ((stack 'top!))
1
> ((stack 'pop!))
```


Definición de un objeto abstracto stack

```
> (define make-stack
      (lambda ( )
        (let ((stk '( )))
             (lambda (message)
                  (case message
                       ((empty?) (lambda ( ) (null? stk)))
                       ((push!) (lambda (x) (set! stk (cons x stk))))
                       ((pop!) (lambda ( ) (if (null? stk)
                                               (error "stack vacio")
                                               (set! stk (cdr stk)))))
                       ((top!) (lambda ( ) (if  (null? stk)
                                               (error "stack vacio")
                                               (car stk))))
                       (else (error "stack: mensaje invalido" message))))))
```


Definición de un stack... ligas compartidas

- El *let* se encuentra dentro del cuerpo de la exp.lambda
- Por lo tanto, se crea un nuevo stack cada vez que se llama a make-stack:

```
> (define s1 (make-stack))
> (define s2 (make-stack))
> ((s1  'push) 1)
> ((s2  'push) 2)
> ((s1 'top))
1
> ((s2 'top))
2
```

Variables protegidas con ligas locales

- Un procedimiento cuyas ligas se hayan modificado tiene un *estado*
- Si se utilizan mensajes para accesar dicho estado, estos procedimientos se conocen como objetos
- Cuando este mecanismo de acceso a variables encapsuladas se combina con el concepto de herencia se tiene la programación orientada a objetos

Reglas de Reducción y Programación Imperativa

- Razonamiento acerca de procedimientos
- El Cálculo Lambda y la conversión-β
- Estrategias de reducción
  - Reducción en orden de aplicación (applicative-order reduction)
  - Reducción por la izquierda
- Definición de procedimientos recursivos en el cálculo lambda
- Secuenciación y programación imperativa
  - Secuenciación
  - Entrada y salida (input-output)
  - Mutación de estructuras de datos
- Asignación y compartición (sharing) de variables
- Corrientes

Corrientes de valores

- Streams: secuencia posiblemente infinita de caracteres que permite el acceso a los valores iniciales, antes que los finale hayan sido generados
- Ejemplo: Los caracteres tecleados en una sesión interactiva
- Frecuentemente es necesario procesar los caracteres iniciales antes que los finales hayan sido tecleados
- Los *stream* se pueden representar como un dato abstracto

Corrientes de valores

- Dato abstracto *stream* (*s* no es vacía)*:*
  - (stream-car s) ;regresa el primer valor
  - (stream-cdr s) :regresa todos los valores menos el primero
  - Es el stream vacío: *the-null-stream*
  - (stream-null? s) ; Indica si el *stream s* es vacío
- Puede haber varias implementaciones

Corrientes de valores

Dato abstracto *stream* (*s* no es vacía)*:*

– (make-stream v p)

make stream regresa un nuevo stream *s*

s = (make-stream v p) ;cons de v y p

tal que:

v = (stream-car s)

p = (stream-cdr s) ;lo que resulta de invocar a *p*

Donde *v* es un valor y *p* es un procedimiento sin argumentos

– *p* es un *thunk*

– se retrasa la formación del resto de la stream hasta que se requiere mediante *stream-cdr*

Corrientes

Una versión de *for-each* para streams:

```
(define stream-for-each
  (lambda (proc stream)
    (letrec ((loop (lambda (stream)
                     (if (not (stream-null? stream))
                         (begin
                           (proc (stream-car stream))
                           (loop (stream-cdr stream)))))))
      (loop stream))))
```

Desplegar una corriente

Esta se puede utilizar para hacer el procedimiento que permite desplegar elemento por elemento del stream y terminar con "newline":

```
> (define display-stream
        (lambda (stream)
                (stream-for-each display stream)
                (newline)))
```


Para convertir un string en un stream de caracteres:

```
(define string->stream
  (lambda (string)
    (let ((string-len (string-length string)))
      (letrec
        ((loop (lambda (cursor)
                 (if (= cursor string-len)
                     the-null-stream
                     (make-stream
                        (string-ref string cursor)
                        (lambda ( ) (loop (+ cursor 1))))))))
          (loop 0)))))
```


Si el *stream* es finito, se puede convertir a una lista que contiene cada uno de sus elementos:

```
(define stream->list
  (lambda (stream)
    (if (stream-null? stream)
        '( )
        (cons (stream-car stream)
              (stream->list (stream-cdr stream))))))
```


El procedimiento (stream-filter pred s)

Toma un predicado *pred* y un stream *s* y produce un nuevo stream que contiene en orden cada uno de los elementos de *s* para los que el predicado *pred* haya resultado verdadero:

```
(define stream-filter
  (lambda (pred stream)
    (cond
      ((stream-null? stream) the-null-stream)
      ((pred (stream-car stream))
           (make-stream
                (stream-car stream)
                (lambda ( ) (stream-filter pred (stream-cdr stream)))))
      (else (stream-filter pred (stream-cdr stream))))))

(define even-positive-integers
        (stream-filter even? positive-integers))
```


Procedimiento para manejar la entrada de caracteres de la terminal como un stream de caracteres.

```
(define make-input-stream
  (lambda ()
    (let ((char (read-char)))
      (if (eof-object? char)
          the-null-stream
          (make-stream char make-input-stream)))))
```


Dentro de un ciclo de lectura e impresión, la construcción del stream de "entrada" se puede utilizar como sigue:

```
(define read-print
  (lambda ()
    (display "--> ")
    (display-stream (make-input-stream))
    (read-print)))
```

¿Cuál sería la apariencia del uso de este ciclo?

```
> (read-print)
--> Una linea de entrada<EOF>
Una linea de entrada
--> esto serian
varias lineas
en la entrada. <EOF>
esto serian
varias lineas
en la entrada
-->
```


Implementación del tipo de dato stream (finita)

Si el stream es finito, se puede representar fácilmente con una lista:

```
> (define stream-car car)
> (define stream-cdr cdr)
> (define make-stream (lambda (value thunk)
                        (cons value (thunk))))
> (define the-null-stream '( ))
> (define stream-null?
        (lambda (stream)
                (eq? stream the-null-stream)))
```


Implementación del tipo de dato stream (no finita)

Si el stream es infinito se utiliza una técnica en la cual se retarda la generacion de los *cdr* tanto como sea posible:

```
> (define stream-cdr
        (lambda (stream) ((cdr stream))))

> (define make-stream
        (lambda (value thunk) (cons value thunk)))

> (define the-null-stream
        (make-stream "end-of-stream"
                (lambda () the-null-stream)))
```


Lazy lists

Normalmente, en la construcción de un cons pair el car y el cdr se evalúan antes de construir la cons-cell; sin embargo, en la construcción de streams, el cdr se evalúa hasta que *stream-cdr* se invoca.

Thunks son una representación finita de una stream posiblemente infinita.

Las *strems* de forma (*valor . thunk*) se conocen como *lazy lists* o listas perezosas!

Si la evaluación del *thunk* tiene efectos laterales, la evaluación "stream-cdr" dos veces provocaría que dichos efectos se duplicaran. Esto se resuelve de la siguiente manera:

```
(define stream-cdr
        (lambda (stream)
                (if (pair? (cdr stream))
                    (cdr stream)
                    (let ((s ((cdr stream))))
                       (set-cdr! stream s)
                       s))))
```


Memoizing o Caching

- La primera vez que un valor se computa se asigna
- Si se requiere después puede ser regresado sin evaluarse
- Comúnmente, el término *stream* se refiere frecuentemente a *streams* en los que la operación *stream-cdr* es *memoizadas* o guardada en una memoria *chache*