Intérpretes

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Intérpretes

- Toman la sintaxis abstracta (parsed) de un programa
- Producen el valor de la expresión
- La especificación del interprete corresponde a la semántica del lenguaje de programación
- Intérprete básico:
  - literales
  - variables
  - aplicaciones
- Estrategía de definición:
  - Se agregan otras formas incrementalmente

Valores expresados y denotados

- Se requiere especificar el conjunto de valores que el lenguaje manipula:
  - Valores expresados (valores posibles o tipos)
    - números
    - pares
    - caracteres
    - strings
  - Valores denotados (valores ligados a variables)
    - celdas que contienen valores expresados
- En el intérprete básico:
  - Valores expresados: Números y procedimientos
  - Valores denotados: Números y procedimientos

Lenguajes de definición y definido

- De definición: lenguaje empleado para definir el lenguaje de programación (meta-lenguaje)
  - En nuestro caso: Scheme
- Definido: lenguaje de programación (objeto)
  - <exp> ::= <integer-literal>
    | <varref>
    | (<operatos><operands>)

  <operatos> ::= <varref> | (<exp>)

  <operand> ::= ( ) | (<operand> {,<operand>}*)

  <operand> ::= <exp>

  <varref> ::= <var>

Expresiones del lenguaje

- 3
- n
- +(3 , n)
- add1(+(3, n))
- (add1)(+(3, n))

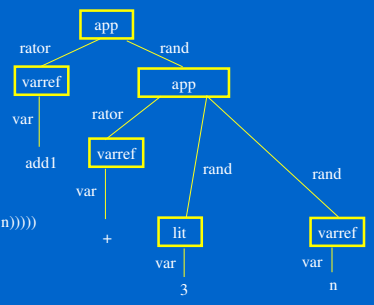
Definición de la sintaxis abstracta

- Definición:
  - (define-record lit (datum))
  - (define-record varref (var))
  - (define-record app (rator rand))

Sintaxis abstracta

add1(+(3, n))

#(app
#(varref add1)
(#(app
#(varref +)
(#(lit 3) #(varref n)))))



Intérprete básico

```
> (define eval-exp
        (lambda (exp)
          (variant-case exp
                (lit (datum) datum)
                (varref (var) (apply-env init-env var))
                (app (rator rands)
                      (let ((proc (eval-exp rator))
                            (args (eval-rands rands))))
                      (apply-proc proc args)))
                (else “error: sintaxis inválida:” exp))))))
> (define eval-rands (lambda (rands) (map eval-exp rands)))
```


apply-env init-env var

- Environment: tipo abstracto función finita
  - ambiente intérprete básico: init-env
  - apply-env regresa el valor denotado por la variable *var* en *init-env*
- Definición de ADT para ambientes:
  - (define the-empty-env (create-empty-ff))
  - (define extend-env extend-ff*))
  - (define apply-env apply-ff)
- Implementación: *ribcage*

Aplicación

- Evaluación en orden aplicativo: Los operadores se llaman antes de llamar al procedimiento (*apply-proc*)
  - (eval-exp rator)
  - (eval-rands rands)
- La aplicación depende de los procedimientos específicos:

<procedure> ::= <prim-op>	prim-proc (prim-op)
<prim-op> ::= <addition>	+
| <substraction>	-
| <multiplication>	*
| <incremement>	add1
| <decrement>	sub1

Aplicación

- Representación de procedimientos primarios:

  > (define-record prim-proc (prim-op))
- Definición de init-env:

```
> (define prim-op-names '(+ - * add1 sub1))
> (define init-env
      (extend-env
            prim-op-names
            (map make-prim-proc prim-op-names)
            the-empty-env))
```


Definición de ADT para procedimientos

```
> (define apply-proc
      (lambda (proc args)
            (variant-case proc
                  (prim-proc  (prim-op) (apply-prim-op prim-op args))
                  (else (error "procedimiento inválido:" proc)))))
> (define apply-prim-op
      (lambda (prim-op args)
            (case prim-op
                  ((+) (+ (car args) (cadr args)))
                  ((-) (- (car args) (cadr args)))
                  ((*) (* (car args) (cadr args)))
                  ((add1) (+ (car args) 1))
                  ((sub1) (- (car args) 1))
                  (else (error "operandor inválido:" prim-op)))))
```


Front-end & parsing

- Primera aproximación:
  - Teclear las expresiones en forma de lista (en vez de usar la sintaxis concreta)

```
> (define run
      (lambda (x)
            (eval-exp (parse x))))
> (run '5)
5
> (run '(add1 2))
3
```


Ciclo de lectura-evaluación-escritura

```
> (define read-eval-print
      (lambda ( )
            (display "-->")
            (write (eval-exp (parse (read))))
            (newline)
            (read-eval-print)))
> (read-eval-print
--> 5                          --> (* (add1 2) (- 6  4))
5                              6
--> (add1 2))
3
```


Front-end & parsing

- Problema: posible confusión entre el lenguaje que está siendo definido y el lenguaje de definición (Scheme).
- Otra aproximación:
  - Definir un parser para leer la entrada como una cadena de caracteres (apéndices D y E):

```
> (define parse character-string-parse)
> (run "5")
5
> (run "add1(2)")
3
```


Front-end & parsing

- Otra aproximación (leyendo la entrada como un *stream*) (según apéndice F) :
  - Definir ciclo read-eval-print para *<exp>*

```
--> 5;
5
--> add1(2);
3
--> *(add1(2),-(6,4));
6
```

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Intérpretes

- Para aumentar propiedades (características) al lenguaje:
  - Se incrementa una producción en la definición de *<exp>*
  - Se especifica la sintaxis abstracta para dicha característica
  - Se incluye el caso correspondiente en el variant-case en el intérprete

Definición de la evaluación condicional

- Se incrementa una producción:
  - Sintaxis concreta:

    <exp> ::= if <exp> then <exp> else <exp>
- Definición para construir la sintaxis abstracta
  - (define-record if (test-exp then-exp else-exp))

Definición de la evaluación condicional

- Se incluye el caso correspondiente en el variant-case en el intérprete:

```
(if (test-exp then-exp else-exp)
    (if (true-value? (eval-exp test-exp))
        (eval-exp then-exp)
        (eval-exp else-exp)))
```


Definición de la evaluación condicional

- Definición de true-value?
  - No hat boleanos como valor expresado por lo que se pueden definir valores de verdad como números: cero es falso y cualquier otro valor es verdadero
  - (define true-value? (lambda (x) (not (zero? x))))

--> if 1 then 2 else 3;

2

--> if - (3, + (1, 2)) then 2 else 3;

x

--> 3

Interprete con condicional

```
> (define eval-exp
    (lambda (exp)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (apply-env init-env var))
        (app (rator rands)
             (let ((proc (eval-exp rator))
                   (args (eval-rands rands))))
             (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
             (if (true-value? (eval-exp test-exp))
                 (eval-exp then-exp)
                 (eval-exp else-exp)))
        (else "error: sintaxis inválida:" exp)))))
```

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Ligas locales con *let*

- Enriquecer el lenguaje con la forma *let*
  - Sintaxis concreta:
    <exp> ::= let <decls> in <exp>
    <decls> ::= <decl> {; <decl>}*
    <decl> ::= <var> = <exp>
  - Ejemplos:

(1) let x = 5; <decls> = <decl; decl>
y = 6 x = 5; y = 6
in +(x, y)

(2) let x = 3 <decls> = <decl> = x = 3
in let x = *(x, x) <decls> = <decl> = x = *(x, x)
in +(x, x)

Ligas locales

- Alcance léxico (lexical scope)
  - La región ligada a la declaración del *let* es el *body*
- Para construir la sintaxis abstracta
  - Un registro para cada declaración *let*
    (define-record let (decls body))
  - Un registro para cada par variable-expresión (cuando menos uno para cada *let*!)
    (define-record decl (var exp))
- Ambiente de evalución
  - Las variables de las declacraciones deben ligarse a su <exp> (RHS)
  - Otras variables deben evaluares en el medioambiente de todo el *let*
  - ergo: el medioambiente de evaluación tiene que ser un parámetro del intérprete

Interprete con "env" como parámetro

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
             (if (true-value? (eval-exp test-exp env)
                 (eval-exp then-exp env)
                 (eval-exp else-exp env)))
        (else "error: sintaxis inválida:" exp)))))
```


Interprete con "env" como parámetro

```
> (define eval-rands
    (lambda (rands env)
      (map (lambda (rand) (eval-exp rand env)
           rands)))
```


Interprete con *let*

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args)))
        ...
```

Interprete con *let* (... cont)

```
(if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env)
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
    (let (decls body)
        (let ((vars (map decl->var decls)
              (exps (map decl->exp decls))
            (let ((new-env (extend-env vars
                                       (eval-rands exps env)
                                       env)))
                (eval-exp body new-env))))
    (else "error: sintaxis inválida" exp))))
```


Lenguajes con alcance léxico

- Una región fija del texto, el *body*, se asocia con las nuevas ligas en un medioambiente
- Si *extend-env* crea una nueva liga para una variable existente, la nueva liga tiene precedencia sobre la anterior. Las declaraciones más anidadas crean hoyos para las externas.

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Procedimientos

- Enriquecer el lenguaje con la forma *proc*
  - Sintaxis concreta:

    <exp> ::= proc <varlist> <exp>

    <varlist> ::= ( ) | (<vars>)

    <vars> ::= <var> {, <var>}*
  - Ejemplos:

    (1) let f = proc (y, z) * (y, - (z, 5)
    in f(2, 8)

    (2) (proc (y, z) *(y, -(z, 5))) (2, 8)); aplicación

Procedimientos

- Enriquecer el lenguaje con la forma *proc*
  - Sintaxis abstracta:

    <exp> ::= proc <varlist> <exp>

    <varlist> ::= ( ) | (<vars>)

    <vars> ::= <var> {, <var>}*
  - Sintaxis abstracta: para definir un procedimiento

    (define-record proc (formals body))
  - Definición de *closure*: para aplicar un procedimiento

Procedimientos

- Ligas y alcance
  - *body* se evalúa en un *environment* en el que los parámetros formales del procedimiento se ligan a los argumentos de la aplicación
  - Reglas de alcance léxico: se deben usar las ligas vigentes al momento de llamar al procedimiento
  - ejemplo:

```
let x = 5
in let f  = proc (y, z) * (y, - (z, x));
     x = 28
     in f(2, 8)
```

    resulta en 6!

Cerradura (closure)

- Un procedimiento puede ser llamado en cualquier parte del código, por lo que hay que guardar su medio-ambiente!
- *Closure*: paquete independiente del ambiente en que se evalúa un procedimiento y que consta de:
  - Cuerpo del procedimiento
  - Lista de parámetros formales
  - Las ligas de las variables libres (environment)
- Definición de "closure"
  - (define-record closure (formals body env))
- Un "closure" se aplica a una lista de argumentos!

Procedimientos no primitivos

- Hay que extender *apply-proc*:

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
        (prim-proc  (prim-op) (apply-prim-op prim-op args))
        (closure (formals body env)
            (eval-exp body (extend-env formals args env)))
        (else (error "procedimiento inválido:" proc)))))
```


Procedimientos

- Hay que extender *apply-proc*:

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
        (prim-proc  (prim-op) (apply-prim-op prim-op args))
        (closure (formals body env)
            (eval-exp body (extend-env formals args env)))
        (else (error "procedimiento inválido:" proc)))))
```


Procedimientos

- Evaluación de *proc*:
  - Crear un closure para el procedimiento
- Extensión del interprete:

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (proc (formals body)
            (make-closure formals body env)))
        ...)))
```


Interprete con *proc*

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (apply-env env var))
        (app (rator rands)
            (let ((proc (eval-exp rator env))
                  (args (eval-rands rands env))))
              (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
            (if (true-value? (eval-exp test-exp env)
                (eval-exp then-exp env)
                (eval-exp else-exp env)))
        ...
```


Interprete con *proc* (... cont)

```
        (let (decls body)
            (let ((vars (map decl->var decls)
                  (exps (map decl->exp decls))
                (let ((new-env (extend-env vars
                                           (eval-rands exps env)
                                           env)))
                  (eval-exp body new-env))))
        (proc (formals body)
            (make-closure formals body env))
        (else "error: sintaxis inválida:" exp)))))
```

Interprete con *proc* (... cont)

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
        (prim-proc  (prim-op) (apply-prim-op prim-op args))
        (closure (formals body env)
                 (eval-exp body (extend-env formals args env)))
        (else (error "procedimiento inválido:" proc)))))
```


Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Asignación de variables

- Estrategía:
  - Modificar ADT extend-env para poder cambiar el valor de una celda
    - Actualmente un *environment* liga cada variable con su valor expresado asociado (*extend-ff*)
    - Para la asignación las variables serán ligadas con localidades de memoria, cuyo contenido es modificado con la asignación
      Denoted Value = Cell (expressed Value)
  - Definir la sintaxis concreta para la asignación
  - Definir la sintaxis abstracta: define record
  - Extender el intérprete

Asignación de variables

- El almacen (the store)
  - La colección de todas las localidades, incluyendo las modificables
  - El valor de las localidades se modica con set-cdr!
- Modificación del intérprete (usando celdas pp. 125)
  - Modificación en *apply-proc*:
    Antes: (extend-env formals args env)
    Ahora: (extend-env formals (map make-cell args) env)
  - Cuando se hace una referencia el valor se tiene que extraer de la celda en la la entrada del *variant-case* correspondiente a *varref:*
    Antes: (varref (var) (apply-env env var))
    Ahora: (varref (var) (cell-ref (apply-env env var)))
- Tipo de referencia: call-by-value

Asignación de variables

- Sintaxis concreta para la asignación:
  <exp> ::= <var> := <exp>
- Registro para interpretación de la sintaxis abstracta:
  (define-record varassign (var exp))
- Implementación de la asignación (aumentar al intérprete en el variant-case)
  - (varassign (var exp)
    (cell-set! (apply-env var) (eval-exp exp env)))
  - *L-values*: Localidades pasadas a *cell-set!*
  - *R-values*: Expressed Values (rhs)

Interprete con *proc*

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (cell-ref (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
               (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
            (if (true-value? (eval-exp test-exp env)
                (eval-exp then-exp env)
                (eval-exp else-exp env)))
        ...
```

Interprete con *proc* (... cont)

```
(let (decls body)
        (let ((vars (map decl->var decls)
              (exps (map decl->exp decls))
          (let ((new-env (extend-env vars
                                     (eval-rands exps env)
                                     env)))
            (eval-exp body new-env))))
(proc (formals body)
        (make-closure fromals body env))
(varassign (var exp)
      (cell-set! (apply-env env var) (eval-exp exp env)))
(else "error: sintaxis inválida:" exp))))))
```


Interprete con *proc* (... cont)

```
> (define apply-proc
   (lambda (proc args)
      (variant-case proc
         (prim-proc  (prim-op) (apply-prim-op prim-op args))
         (closure (formals body env)
                  (eval-exp
                        body
                        (extend-env
                              formals
                              (map make-cell args)
                              env)))
         (else (error "procedimiento inválido:" proc)))))
```


Secuenciación

- Sintaxis concreta:

  <exp> ::= begin <exp> <compound>

  <compound> ::= {;<exp>}* end
- Sintaxis abstracta:

  (define-record begin (exp1 exp2))
- Ejemplo:

```
--> let x = 3
        in begin
                x := add1(x);
                x := +(x x);
                +(x, 2)
        end;
10
```


Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Tres formas de definir la recursión

- Utilizando el combinador Y en el cálculo lambda
  - Interesante desde el punto de vista teórico
  - Se podría implementar directamente en el intérprete
  - Ineficiente desde del punto de vista práctico
- Utilizando el *letrec*
  - Se crean cerraduras para cada llamada recursiva
  - Los *environment* contienen ligas circulares
- Creando una forma especial para manejar la recursión directamente sin circularidad

Por transformación de *letrec*

- Se transforma sintácticamente una expresión *letrec* en una expresión usa un *let* para crear un *environment* con ligas auxiliares a variables (dummy variable bindings)
- Los procedimientos recursivos son cerrados en dicho *environment*
- Se utiliza la asignación para incluir los *closures* en las celdas asociadas a las variables auxiliares

Trasnformación del *letrec*

(letrec ((var_1 exp_1) ... (var_n exp_n))
  body)

⇒

```
(let (($var_1$ '*) ... ($var_n$ '*))                ;ligas auxiliares
  (let (($v_1$ $exp_1$) ... ($v_n$ $exp_n$))        ;se evaluan los proc.
    (set! $var_1$ $v_1$)                             recursivos y se asignan
    ...                                              a las variables dummy
    (set! $var_n$ $v_n$)
    'unspecified
  body)
```


Ejemplo

(letrec ((var_1 exp_1))
  body)

⇒

```
(let (($var_1$ '*))
  (let (($v_1$ $exp_1$))
    (set! $var_1$ $v_1$)
    'unspecified
  body)
```

```
letrec((fact (lambda (n) ...)
  ...)

⇒

(let ((fact '*))
  (let (($v_1$ (lambda (n) .... ))
    (set! fact $v_1$)
    'unspecified
  body)
```


Recursión directa en el interprete sin circularidad

- Enriquecer la sintaxis con una variante de *letrec*, restringiendo el RHS de las declaraciones recursivas a expresiones *proc*
- Sintaxis concreta:
  - <exp> ::= letrecproc <procdecls> in <exp>
  - <procdecls> ::= <procdecl> {; <procdecl>}*
  - <procdecl> ::= <var> <varlist> = <exp>
- Para definir sintaxis abstracta
  - (define-record letrecproc (procdecls body))
  - (define-record procdecl (var formals body))

Recursión directa en el interprete sin circularidad

- Ejemplos:

```
letrecproc
        fact(x) = if zero(x) then 1 else *(x, fact(sub1(x)))
in fact(6)

letrecproc
        even(x) = if zero(x) then 1 else odd(sub1(x));
        odd(x) = if zero(x) then 0 else even(sub1(x))
in odd(13)
```


Agregar *letrecproc* al intérprete

- Definir *extend-rec-env*
  - operador para el ADT *env* extiende el *environment* con un conjunto the procedimientos mutuamente recursivos
- Modificar el *variant-case* de *eval-exp* para incorporar el nuevo tipo de registro en la sintaxis abstracta (*letrecproc*).
- La nueva entrada es similar a *let* pero utiliza *extend-rec-env* en vez del *extend-env*
- Implementación:
  - Se almacena la información del *closure* de una llamada recursiva, excepto en *env* en un registro *proc*
  - Se define un *closure* completo antes de regresar un valor

Environment ADT sin circularidad

```
(define-record extended-rec-env (vars vals old-env))

(define extend-rec-env                  ;utilizando ribassoc
        (lambda (procdecls env)
          (make-extended-rec-env
                (map procdecl->var procdecls)
                (list->vector
                        (map (lambda (procdecl)
                                (make-proc
                                        (procdecl->formals procdecl)
                                        (procdecl->body procdecl)))
                        procdecls))
                env)))
```

Environment ADT sin circularidad

```
( (define apply-env                    ;utilizando ribassoc
        (lambda (env var)
          (variant-case env
                (extended-rec-env (vars vals old-env)
                    (let ((p (ribassoc var vars vals '*fail*)))
                        (if (eq? p '*fail*)
                            (apply-env old-env var)
                            (make-closure (proc->formals p)
                                          (proc->body p)
                                          env))))
                ... el tipo normal de environment)))
```


Interprete con *letrecproc*

```
> (define eval-exp
      (lambda (exp env)
        (variant-case exp
            (lit (datum) datum)
            (varref (var) (apply-env env var))
            (app (rator rands)
                 (let ((proc (eval-exp rator env))
                       (args (eval-rands rands env))))
                 (apply-proc proc args)))
            (if (test-exp then-exp else-exp)
                 (if (true-value? (eval-exp test-exp env)
                     (eval-exp then-exp env)
                     (eval-exp else-exp env)))
            ...
```


Interprete con *letrecproc* (... cont)

```
(let (decls body)
        (let ((vars (map decl->var decls)
              (exps (map decl->exp decls))
            (let ((new-env (extend-env vars
                                       (eval-rands exps env)
                                       env)))
                (eval-exp body new-env))))
(proc (formals body)
        (make-closure formals body env))
(letrecproc (procdecls body)
        (eval-exp body (extend-rec-env procdecls env)
(else "error: sintaxis inválida:" exp)))))
```


Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Alcance y asignación dinámica

- Alcance dinámico
  - Mecanismo de liga de variables (binding) muy expresivo
  - Difícil razonar acerca de programas con ligas con alcance dinámico
- Asignación dinámica
  - Ventajas de las ligas con alcance dinámico
  - Simplicidad de las ligas con alcance léxico

Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Alcance dinámico

- Alcance léxico
  - Los procedimientos se evalúan en el *environment* de creación
  - Se requiere un nuevo *closure* cada vez que una expresión "proc" es evaluada
- Alcance dinámico
  - Los procedimientos se evalúan en el *environment* vigente al momento de evaluación (no de creación)
  - Una liga dinámica es vigente sólo durante la evaluación del *body* asociado con la liga
  - Las variables en el cuerpo del procedimiento utilizan las ligas que se ha establecido más recientemente en el proceso de evaluación

Alcance léxico versus dinámico

- Alcance léxico

```
> let a = 3
  in let  p = proc (x) +(x, a);
          a = 5
     in *(a, p(2))
⇒    *(5, +(2, 3)) = 25
```

- Alcance dinámico

```
> let a = 3
  in let  p = proc (x) +(x, a);
          a = 5
     in *(a, p(2))
⇒    *(5, +(2, 5)) = 35
```


Alcance dinámico

- Los procedimientos no se cierran sobre su ambiente de creación
- Sólo se requiere entonces almacenar la lista de parámetros formales y el cuerpo del procedimiento!
- Consecuentemente, la cláusula de *proc* en el *variant-case* de *eval-exp* es:

```
(proc (formal body) exp)
```

- La evaluación es simple y eficiente

Intérprete con *proc* (liga léxica)

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (cell-ref (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
             (if (true-value? (eval-exp test-exp env)
                 (eval-exp then-exp env)
                 (eval-exp else-exp env)))
        ...
```


Intérprete con *proc* (liga léxica)

```
(let (decls body)
     (let ((vars (map decl->var decls)
           (exps (map decl->exp decls))
          (let ((new-env (extend-env vars
                                     (eval-rands exps env)
                                     env)))
               (eval-exp body new-env))))
(proc (formals body)
      (make-closure fromals body env))
(varassign (var exp)
     (cell-set! (apply-env var) (eval-exp exp env)))
(else "error: sintaxis inválida:" exp)))))
```


apply-proc (liga léxica)

```
> (define apply-proc
   (lambda (proc args)
     (variant-case proc
       (prim-proc  (prim-op) (apply-prim-op prim-op args))
       (closure (formals body env)
                (eval-exp
                     body
                     (extend-env
                          formals
                          (map make-cell args)
                          env)))
       (else (error "procedimiento inválido:" proc)))))
```

Intérprete con *proc* (liga dinámica)

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (cell-ref (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args env)))     ; se pasa el env!
        (if (test-exp then-exp else-exp)
            (if (true-value? (eval-exp test-exp env)
                (eval-exp then-exp env)
                (eval-exp else-exp env)))
        ...
```


Intérprete con *proc* (liga dinámica)

```
(let (decls body)
     (let ((vars (map decl->var decls)
           (exps (map decl->exp decls))
           (let ((new-env (extend-env vars
                                      (eval-rands exps env)
                                      env)))
                (eval-exp body new-env))))
(proc (formals body) exp)  ;se utiliza en env vigente!
(varassign (var exp)
     (cell-set! (apply-env var) (eval-exp exp env)))
(else "error: sintaxis inválida:" exp))))
```


apply-proc (liga léxica)

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
        (prim-proc  (prim-op) (apply-prim-op prim-op args))
        (closure (formals body env)
                 (eval-exp
                      body
                      (extend-env
                           formals
                           (map make-cell args)
                           env)))
        (else (error "procedimiento inválido:" proc)))))
```


apply-proc (liga dinámica)

```
> (define apply-proc
    (lambda (proc args current-env)
      (variant-case proc
        (prim-proc  (prim-op) (apply-prim-op prim-op args))
        (proc (formals body)
                 (eval-exp
                      body
                      (extend-env
                           formals
                           (map make-cell args)
                           current-env)))
        (else (error "procedimiento inválido:" proc)))))
```


Alcance dinámico & stack environment

- Con alcance dinámico, cuando una liga se adiciona al ambiente de evaluación permanece en efecto hasta que se remueve, por lo que el ambiente obedece una disciplina de stack (last-in-first-out)
- Por lo tanto, es posible diseñar el intérprete con un stack global (en vez de pasar el *environment* como un parámetro)

Intérprete con *env*

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (cell-ref (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args env)))
        (if (test-exp then-exp else-exp)
            (if (true-value? (eval-exp test-exp env)
                (eval-exp then-exp env)
                (eval-exp else-exp env)))
        ...
```

Intérprete con *env*

```
(let (decls body)
        (let ((vars (map decl->var decls)
              (exps (map decl->exp decls))
          (let ((new-env (extend-env vars
                                     (eval-rands exps env)
                                     env)))
            (eval-exp body new-env))))
(proc (formals body) exp)
(varassign (var exp)
        (cell-set! (apply-env var) (eval-exp exp env)))
(else "error: sintaxis inválida:" exp)))))
```


Intérprete con *stack* global

```
> (define eval-exp
        (lambda (exp)
          (variant-case exp
                (lit (datum) datum)
                (varref (var) (cell-ref (lookup-in-env var))
                (app (rator rands)
                        (let ((proc (eval-exp rator))
                              (args (eval-rands rands))))
                        (apply-proc proc args)))
                (if (test-exp then-exp else-exp)
                        (if (true-value? (eval-exp test-exp)
                            (eval-exp then-exp)
                            (eval-exp else-exp)))
                ...
```


Intérprete con *stack* global

```
(proc (formals body) exp)
(varassign (var exp)
        (cell-set! (look-up-in-env var) (eval-exp exp)))
(else "error: sintaxis inválida:" exp)))))

(define eval-rands
  (lambda (rands)                    ;sin env!
        (map (lambda (exp) (eval-exp exp))
             rands)
```


apply-proc (stack global)

```
> (define apply-proc
    (lambda (proc args)
      (variant-case proc
            (prim-proc  (prim-op) (apply-prim-op prim-op args))
            (proc (formals body)
                  (push-env! formals (map make-cell args))
                  (let ((value (eval-exp body)))
                    (pop-env!)
                   value))
            (else (error "procedimiento inválido:" proc)))))
```


Intérpretes

- Un interprete simple (básico)
- Evaluación condicional
- Ligas locales
- Procedimientos
- Asignación de variables
- Recursión
- Alcance y asignación dinámica
  - Alcance dinámico
  - Asignación dinámica

Asignación dinámica

- Alcance léxico: las ligas tienen vida indefinida (siempre que se necesiten)
- Se mantienen permanentemente en el *closure*
- Alcance dinámico: ligas con "vida" finita
- Asignación dinámica (fluid binding):
  - asignación dinámica a ligas con alcance léxico

Asignación dinámica

- Ejemplo:

```
let x = 4
 in let p = proc (y) +(x, y)
    in +(x := 7 during p(1), p(2))
```

para p(1) x = 7
para p(2) x = 4

⇒ +((7, 1), (4, 2)) = 14

Asignación dinámica

- Sintaxis concreta para la asignación:
  <exp> ::= <var> := <exp> during <exp>
- Registro para interpretación de la sintaxis abstracta:
  (define-record dynassign (var exp body))
  que codifica:
  var := *exp* during *body*
- ejemplo:

```
let x = 4
 in let p = proc (y) +(x, y)
    in +(x := 7 during p(1), p(2))
```


Interprete con *dynassign*

```
> (define eval-exp
    (lambda (exp env)
      (variant-case exp
        (lit (datum) datum)
        (varref (var) (cell-ref (apply-env env var))
        (app (rator rands)
             (let ((proc (eval-exp rator env))
                   (args (eval-rands rands env))))
             (apply-proc proc args)))
        (if (test-exp then-exp else-exp)
             (if (true-value? (eval-exp test-exp env)
                 (eval-exp then-exp env)
                 (eval-exp else-exp env)))
        ...
```


Interprete con *dynassign* (... cont)

```
(let ...
(proc (formals body)
        (make-closure fromals body env))
(varassign (var exp)
     (cell-set! (apply-env var) (eval-exp exp env)))
(dynassign (var exp body)
        (let ((a-cell (apply-env env var))
              (b-cell (make-cell (eval-exp exp env))))
        (cell-swap! a-cell b-cell)
        (let ((value (eval-exp body env)))
             (cell-swap! a-cell b-cell)
           value)))
(else "error: sintaxis inválida:" exp)))))
```