Abstracción Sintáctica y Abstracción de Datos

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Ligas en Scheme

- Globales
  - a nivel del intérprete
- Locales
  - mediante procedimientos lambda
- Ligas para uso inmediato
  - let
  - letrec

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Ligas locales: let

```
> (define subst
    (lambda (new old slst)
          (if (null? slst)
              '( )
              (if (symbol? (car slst))
                  (if  (eq? (car slst) old)
                       (cons new  (subst new old (cdr slst)))
                       (cons (car slst) (subst new old (cdr slst))))
                  (cons (subst new old (car slst))
                        (subst new old (cdr slst)))))))
```

• Tres veces...

Ligas locales: let

```
> ((lambda (cdr-result)
          (if (symbol? (car slst))
              (if  (eq? (car slst) old)
                   (cons new cdr-result)
                   (cons (car slst) cdr-result))
              (cons (subst new old (car slst))
                    cdr-result)))
    (subst new old (cdr slst)))
```

• Se ligan los argumentos formales
• Se evalúa el cuerpo de la función lambda
• *(car slst)* se evalúa tres o cuatro veces!

Ligas locales: let

```
> ((lambda (car-value  cdr-result)
          (if (symbol? car-value)
              (if  (eq? car-value old)
                   (cons new cdr-result)
                   (cons car-value cdr-result))
              (cons (subst new old car-value)
                    cdr-result)))
    (car slst)
    (subst new old (cdr slst)))
```

• Se ligan los argumentos formales
• Se evalúa el cuerpo de la función lambda
• Se reduce el esfuerzo computacional... difícil de leer!

Ligas locales: let

```
> (define subst
    (lambda (new old slst)
        (if  (null? slst)
             '( )
             (let ((car-value (car slst))
                   (cdr-result (subst new old (cdr slst))))
               (if (symbol? car-value)
                   (if  (eq? car-value old)
                        (cons new cdr-result)
                        (cons car-value cdr-result))
                   (cons (subst new old car-value)
                         cdr-result))))))
```


Ligas locales: let

• let

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

La región asociada a var_1,..., var_n es *body*

• Lo cual es equivalente a:

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

• ergo: azucar sintáctico

Ligas locales: let

• Alcance de let:

```
(let  ((x 3))
   (let ((y (+ x 4)))          ;body
        (* x y))
)
```

• equivalente a:

```
((lambda (x)
        ((lambda (y) (* x y)) (+ x 4))
   )
 3)
```

reduciendo: ((lambda (y) (* 3 y)) (+ 3 4))
reduciendo: (* 3 7)

Ligas locales: let

• Forma *let*:

```
(let  ((x 3)
       (y (+ x 4))
       )
  (* x y))
)
```

• equivalente a:

```
((lambda (x y) (* x y))
  3                    ;1er. argumento
  (+ x 4)              ;2o. argumento
 )
```

• x debe ligarse globalmente:

```
(*  3 (+ x 4))
```


Ligas locales: let

• Alcance de let:

```
(let  ((x 3))
   (let ((x (* x x)))
         (+ x x))
)
```

• equivalente a:

```
((lambda (x)
        ((lambda (x) (+ x x)) (* x x))
 3)
```

• Notar que solo x en (* x x) es libre y sujeta a substitución:

reduciendo: ((lambda (x) (+ x x)) (* 3 3))

reduciendo: (+ 9 9)

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Ligas locales: letrec

```
> (define partial-vector-sum
    (lambda (von n)
          (if (zero? n)
             0
             (+ (vector-ref von (- n 1))
                (partial-vector-sum von (- n 1))))))

> (define vector-sum
    (lambda (von)
          (partial-vector-sum von  (vector-length von))))
```

• Dos procedimientos para una operación local!

Ligas locales: letrec

• Sugerencia?:

```
> (let ((partial-vector-sum
                (lambda (von n)
                   (if (zero? n)
                      0
                      (+ (vector-ref von (- n 1))
                         (partial-vector-sum von (- n 1)))))))
          (partial-vector-sum von  (vector-length von))))
```

• El alcande de los símbolos en *let* es el *body*

Ligas locales: letrec

• Veámoslo más claramente:

```
> (let ((partial-vector-sum
                (lambda (von n)
                   ...  (partial-vector-sum von (- n 1))...)))
          (partial-vector-sum von  (vector-length von))))
```

• Equivalente a (o algo así!):

```
((lambda (partial-vector-sum)
          (partial-vector-sum von  (vector-length von)))
                (lambda (von n)
                   ...  (partial-vector-sum von (- n 1))
```

Ligas locales: letrec

• letrec

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

• La región asociada a var_1,..., var_n:
toda la expresión letrec y no sólo el *body*

• Consecuentemente exp_1,..., exp_n pueden contener procedimientos mutuamente recursivos

Ligas locales: letrec

• Restricción: las varibles pueden ser ligadas sólo cuando sus expresiones asociadas han sido evaluadas

```
(letrec ((x 3))
        (y (+ x 1)))
   y)
```

• Esto se puede resolver mediante el uso de expresiones lambda, ya que las variables en el cuerpo de un procedimiento lamba se evalúan cuando el procedimiento resultante es evaluado, y no cuando la expresión lambda se invoca.

```
(letrec ((x 3))
        (y (lambda ( )  (+ x 1)))
   (y))
```


Ligas locales: letrec

• Ahora sí!

```
> (define vector-sum
    (lambda (von)
      (letrec ((partial-vector-sum
                  (lambda (n)
                    (if (zero? n)
                        0
                        (+ (vector-ref von (- n 1))
                           (partial-vector-sum (- n 1)))))))
        (partial-vector-sum (vector-length von)))))
```


Ligas locales: letrec

• Otro ejemplo:

```
>  (letrec ((even? (lambda (n)
                     (if  (zero? n)
                          #t
                          (odd? (- n 1))))
                (odd? (lambda (n)
                     (if  (zero? n)
                          #f
                          (even? (- n 1))))))
   (even? 3)
```


Ventajas de Ligas locales

- Es más fácil trabajar con definiciones locales que con globales
- El efecto de modificar el código se limita al alcance de la declaración local
- Frecuentemente se reduce el número de argumentos requeridos, ya que algunas ligas están en el contexto
- Se reduce el número de definiciones globales y se previenen conflictos de nombres de símbolos, especialmente en programas grandes

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Conectivos Lógicos

- Conjunción (and) & disjunción (or)
- Se pueden definir como proc. (los operandos se evalúan antes de llamar al operador).
- En Scheme se evalúan de izquierda a derecha (formas especiales) la función adquiere valor tan pronto como sea posible:
  - *and*: primer argumento falso = falso
  - *or*: primer argumento verdadero = verdad
- Cualquier valor diferente de #f es tratado como verdadero
  - (and 3 (number? #t) --> #f
  - (and #t (number? 3) --> #t
  - (or (number? #t) 3 4) --> 3

Conectivos Lógicos

- Especificación inductiva:
  (and $test_1$ $test_2$ ... $test_n$)
  es equivalente a:
  (if $test_1$
  (and $test_2$ ... $test_n$)
  #f)
- De manera similar (or $test_1$ $test_2$ ... $test_n$)
  (if $test_1$
  #t
  (or $test_2$ ... $test_n$))
- Alternativamente para regresare el valor de *test* sin volverlo a evaluar:
  (let ((*value* $test_1$))
  (if *value*
  value
  (or $test_2$ ... $test_n$))))

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Cond

- Selector de trayectorias:
  (cond
  ($test_1$ $consequent_1$)
  ...
  ($test_n$ $consequent_n$)
  (**else** *alternative*))
- Se evalúan los *test* de manera secuencial
- Se ejecuta el consecuente asociado al primer test que resulte verdadero.
- Azucar sintáctico:
  - **cond** se puede reescribir como una lista de ifs anidados

Cond

```
> (define subst
    (lambda (new old slst)
            (cond
              ((null? slst)  '( ))
              ((symbol? (car slst))
                 (if  (eq? (car slst) old)
                      (cons new  (subst new old (cdr slst)))
                      (cons (car slst) (subst new old (cdr slst))))
                 (else cons (subst new old (car slst))
                           (subst new old (cdr slst)))))))
```


Case

- Selector de trayectorias comparando con un valor de referencia:
  (case *key*
  ($key\text{-}list_1$ $consequent_1$)
  ...
  ($key\text{-}list_n$ $consequent_n$)
  (**else** *alternative*))
- $key\text{-}list_i$ es una lista de símbolos, números, boleanos o caracteres
- La expresión *key* se evalúa y su valor se compara con los $key\text{-}list_i$
- Se ejecuta el consecuente asociado al primer match

Transformación *case* a *let* y *cond*

• Azucar sintático:

```
(let ((*key*  key)
        (cond
                (memv ((*key*  'key-list1 ) consequent1)
                ...
                 (memv ((*key*  'key-listn ) consequentn)
                (else  alternative)))
```

- **key** no ocurre libre en los consecuentes o la alternativa
- **memv** toma una llave y una lista y es verdad si la llave está en la lista

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Registros (Records)

- Estructuras de datos cuyos elementos se accesan por nombre (en vez de por posición como pares doteados o vectores).
- Cada registro tiene un *nombre* y un conjunto de campos.

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Dr. Luis A. Pineda, IIMAS, UNAM, 2000.

Definición de Registros

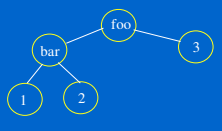
- define-record:
  - (define-record *name* (*field*$_1$ ... *field*$_n$))
- Se define:
  - Registros de tipo ***name***
  - Un predicado ***name?*** (verdad si se le pasa un registro de tipo *name*)
  - Un procedimiento ***make-name*** que toma *n* argumentos y regresa un registro del tipo con el argumento i-esimo en el campo *field*$_i$
  - Un conjunto de procedimientos de acceso ***name->field***$_i$ para *1 <= i <= n*

Ejemplo: árbol-interior

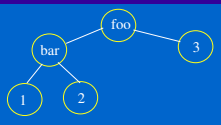
- <tree> ::= <number> | (<symbol> <tree> <tree>)

```
> (define-record interior (symbol left-tree right-tree))
> (define tree-1 (make-interior 'foo (make-interior 'bar 1 2) 3))

> (interior? tree-1)
> #t
> (interior->symbol tree-1)
> foo
> (interior->right-tree (interior->left-tree tree-1))
> 2
```



Ejemplo: árbol-interior




```
> (define leaf-sum
    (lambda (tree)
      (cond
          ((number? tree) tree)
          ((interior? tree) (+ (leaf-sum (interior->left-tree tree))
                               (leaf-sum (interior->right-tree tree))))
          (else (error  "leaf-sum: arbol invalido")))))
> (leaf-sum tree-1)
> 6
```


Resumiendo...

- Un tipo de dato distinto puede utilizarse para representar cada alternativa de una especificación BNF
- Cada símbolo no terminal en la alternativa se representa como un campo en su registro asociado
- Para el caso de árboles:
  - <tree> ::= <number> | (<symbol> <tree> <tree>)
- Tenemos:
  - (define-record interior (symbol left-tree right-tree))
  - (define-record leaf (number))

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Variant records & variant-case

- Tipo unión: combina dos o más tipos se le llama *tipo unión* (*union-type*)
- El tipo tree es el tipo unión de:
  - interior
  - leaf
- variant record: tipo unión tal que todas sus alternativas son registros.

Variant-case

- Permite seleccionar entre un tipo de registro
- Permite ligar sus campos a variable nombradas de acuerdo con los campos

  (variant-case *record-expression*
  (*name*$_1$ *field-list*$_1$ *consequent*$_1$)
  ...
  (*name*$_n$ *field-list*$_n$ *consequent*$_n$)
  (else *alternative*))
- donde *fiel-list*$_i$ es la lista de campos del tipo *name*$_i$
- se evalúa *record-expression* y si su valor *v* es de algún tipo *name*$_i$ sus valores se ligan con los valores correspondientes de *field-list*$_i$; después se evalúa *consequent*$_i$

Variant-case

- Ejemplo: leaf-sum

```
(define leaf-sum
  (lambda (tree)
    (variant-case tree
      (leaf  (number) number)
      (interior (left-tree right-tree)
            (+ (leaf-sum left-tree) (leaf-sum right-tree)))
      (else "error: leaf-tree: árbol invalido"))))
```

Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Sintáxis abstracta y su representación mediante registros

- Sintaxis concreta:
  - el programa como una cadena de caracteres
- Sintaxis abstracta:
  - La estructura interna del programa
  - La forma que *dirige* al interprete o al compilador
- Ejemplo:
  - sintaxis concreta: (lambda (x) (f (f x)))
  - sintaxis abstracta (árbol generado por):

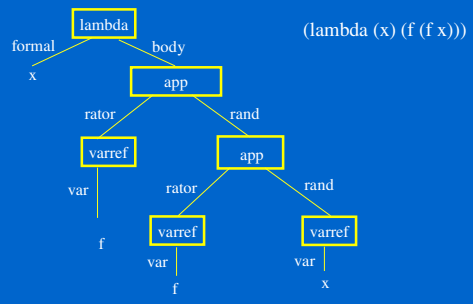
    <exp> ::= (lamba (<var>) <exp>)

Diseño de sintaxis abstracta

- Nombrar:
  - Cada producción de la sintaxis BNF
  - Cada ocurrencia de un símbolo no-terminal en cada producción
- Para el cálculo lambda:

Producción	Nombre
– <exp> ::= <number>	lit (num)
| <var>	varref (var)
| (lambda (<var>) <exp>)	lambda (formal body)
| (<exp> <exp>)	app (rator rand)

Sintaxis abstracta (representada mediante variant-records)



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

Sintaxis abstracta

- La sintaxis abstracta se puede definir con registros:

  > (define-record lit (datum))

  > (define-record varref (var))

  > (define-record lambda (formal body))

  > (define-record app (rator rand))

Parser & Sintaxis abstracta

```
> (define parse
    (lambda (datum)
      (cond
        ((number? datum) (make-lit datum))
        ((symbol? datum) (make-varref datum))
        ((pair? datum)
          (if (eq? (car datum) 'lambda)
                (make-lambda (caadr datum) (parse (caddr datum)))
                (make-app (parse (car datum)) (parse (cadr datum)))))
          (else (error "parser: error gramatical" datum)))))
```

des-parser & Sintaxis abstracta

```
> (define unparse
    (lambda (exp)
      (variant-case exp
        (lit (datum) datum)
        (varref (var)  var)
        (lambda (formal body)
          (if (eq? (car datum) 'lambda)
                (list 'lambda (list formal) (unparse body)))
        (app (raptor rand) (list (unparse rator) (unparse rand)))
         (else (error "unparser: sintaxis abstracta invalida" exp)))))
```


Otro ejemplo: free-vars

```
> (define free-vars
    (lambda (exp)
      (variant-case exp
        (lit (datum) '())
        (varref (var)  (list var))
        (lambda (formal body) (remove formal (free-vars body))
        (app (raptor rand) (union (free-vars rator) (free-vars rand))))))
```


Abstracción Sintáctica y de Datos (1)

- Ligas locales
  - let
  - letrec
- Conectivos Lógicos
- Selección (Branching)
  - cond
  - case
- Registros (Records)
  - define-record
  - variant Records y variant-case
  - Sintáxis abstracta y su representación mediante registros
  - Una implementación de registros

Una implementación de registros

- Los registros son abstractos (independientes de implementación):
  - Con vectores
  - Con listas
  - Con árboles binarios
  - Se puede proveer por Scheme
- Implementación con vectores
  - Eficiente en tiempo de acceso y recursos de memoria
- Ejemplo:

```
> (parse '(lambda (x) (y x)))
> #(lambda x #(app #(varref y)  #(varref x)))
```

formato: #(tipo $campo_1$ $campo_2$...)

Una implementación de registros

- Definición (ejemplo):

```
> (define-record leaf (number))
```

- Procedimiento *make-name*

```
> (define make-leaf
      (lambda (number)
            (vector  'leaf  number)))
```


Una implementación de registros

- Predicado *name?*

(se tiene que analizar la estructura del tipo de dato)

```
> (define leaf?
    (lambda (obj)
          (and (vector? obj)
                (= (vector-length obj) 2)
                (eq? (vector ref obj 0) '(leaf))))
```

- Selector:

```
> (define leaf-number
    (lambda (obj)
      (if (leaf? obj)                    ;se hace uso del predicado
          (vector-ref obj 1)             ; se extrae el valor
          (error "leaf-number: registro invalido" obj))))
```

Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Abstracción de datos

- Se abstrae sobre los detalles de la implementación
  - Una tabla se puede representar como un árbol binario balanceado
- Se puede cambiar la implementación del tipo de datos
  - Sólo se afectan los procedimientos de interfaz (make, name? y selectores)
- Utilidad en creación de prototipos
  - desarrollar con tabla y depurar con un árbol binario balanceado
- Técnicas de abstracción de datos (ADT techniques):
  - Crear un tipo de datos abstracto y un conjunto de procedimientos para accesar, de manera exclusiva, dichos tipos de datos.
  - El resto del programa es independiente de la implementación
- ADT pueden ser especificados de manera algebraica
  - Sistemas de ecuaciones

Abstracción de datos

- Ejemplo de abstracción de datos:
  - define-record
- Problema:
  - Nada impide que los datos sean accesados directamente por otros procedimientos (i.e., *vector-ref*)
  - Tipos disjuntos (disjoints types): datos de un solo tipo
- Ejemplos:

```
> (interior? tree-1)
> #t
> (vector? tree-1)
> #t
> (vector-ref tree-1 0)
> interior
```


Abstracción de datos

- Tipos ***opacos*** *versus* ***transparentes***
  - No pueden exponerse por medio de ninguna operación (incluso impresión)
- Los tipos primitivos en Scheme (procedure, number, pair, vector, character, symbol, string, boolean y empty list) son:
  - Mutuamente disjuntos
  - opacos
- List es un tipo derivado no abstracto (comparte proc. de creación y selección con *pair*)
- Empty list es un "singleton type" con un solo elemento y el predicado *null?*
- No hay mecanismos en Scheme para crear tipos opacos; hay que adoptar las convenciones de *define-record*

Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Representaciones procedurales versus representaciones estructuradas

- Representaciones procedurales
  - Opacas
  - Semántica implícita
  - Procedimientos de primer orden
- Representaciones estructuradas
  - Más eficientes?
  - Semántica declarativa
  - Pueden ser implementadas en lenguajes de bajo nivel

Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Representaciones procedurales

- Consideremos el concepto de *función finita*
  - medioambientes de programación (asocia variables con valores)
  - Tablas de símbolos (asocia variables con su dirección léxica)
- Representaciones: de un conjunto de pares versus de una función
  - Conjunto de pares: {(d, 6), (x, 7) (y, 8)}
  - función: (lambda (arg)
    (case arg
    ((d) 9) ((x) 7) ((y) 8)
    (else (error "argumento inválido"))))
- Tipo de dato abstracto *función finita:*
  - create-empty-ff: crea una funciín finita vacía
  - extend-ff: extiende una función finita, de tipo función finita
  - apply-ff: aplica la función finita

Representaciones procedurales

- Construcción y acceso del tipo *ff*:

```
> (define dxy-ff
        (extend-ff 'd 6
                (extend-ff 'x 7
                        (extend-ff 'y 8
                                (create-empty-ff))))

> (apply-ff dxy-ff 'x)
7
```


Representaciones procedurales

- Interface al tipo de datos *ff*:

```
> (define create-empty-ff
    (lambda ( )
        (lambda (symbol)
            (error "función vacía: no hay valor para: symbol))))
> (define extend-ff
    (lambda (sym val ff)
        (lambda (symbol)
            (if (eq? symbol sym)
                val
                (apply-ff ff symbol)))))
> (define apply-ff
    (lambda (ff symbol)
        (ff symbol)))
```


Representaciones procedurales

- Construcción y acceso del tipo *ff*:

```
> (define dxy-ff
        (extend-ff 'd 6
                (extend-ff 'x 7
                        (extend-ff 'y 8
                                (create-empty-ff))))

> (apply-ff dxy-ff 'x)
7
```


Representaciones procedurales

```
> (apply-ff dxy-ff 'x)
⇒ (dxy-ff x)))
⇒ ((extend-ff 'd 6
        (extend-ff 'x 7
                (extend-ff 'y 8 (create-empty-ff)))) x)

⇒  (lambda (symbol)
        (if (eq? symbol d)
            6
            (apply-ff (extend-ff 'x 7
                        (extend-ff 'y 8 (create-empty-ff)))) symbol)))
    x)
```

Representaciones procedurales

```
⇒ (if (eq? x d)
      6
      (apply-ff (extend-ff 'x 7
                        (extend-ff 'y 8 (create-empty-ff)))) x)
⇒ ((extend-ff 'x 7
              (extend-ff 'y 8 (create-empty-ff)))) x)

⇒ (lambda (symbol)
            (if (eq? symbol x)
                7
                (apply-ff (extend-ff 'y 8 (create-empty-ff)))) symbol)))
   x)
```


Representaciones procedurales

```
⇒ (if (eq? x x)
      7
      (apply-ff (extend-ff 'y 8 (create-empty-ff)) x))

⇒ 7
```


Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Representación de registros (Record Representation)

- Representaciones procedurales pueden ser transformadas como estructuras declarativas; para las funciones finitas tenemos:

```
> (define-record empty-ff ( ))
> (define-record extended-ff (sym val ff))
```

- Procedimientos de acceso al tipo finite-function:

```
> (define create-empty-ff
          (lambda ( ) (make-empty-ff)))
```

Alternativamente:

```
> (define create-empty-ff make-empty-ff)
```


Representación de registros (Record Representation)

- Procedimientos de acceso al tipo finite-function:

```
> (define extend-ff
          (lambda (sym val ff) (make-extended-ff sym val ff)))
```

- Alternativamente:

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


Representación de registros (Record Representation)

```
> (define apply-ff
    (lambda (ff symbol)
      (variant-case ff
        (empty-ff ( ) (error "empty-ff: no hay simbolo" symbol))
        (extended-ff (sym val ff)
                     (if (eq? symbol sym)
                         val
                         (apply-ff ff symbol))))
        (else (error "apply-ff: función finita inválida" ff)))))
```

Representaciones procedurales

- Construcción y acceso del tipo *ff*:

```
> (define dxy-ff
        (extend-ff 'd 6
                (extend-ff 'x 7
                        (extend-ff 'y 8
                                (create-empty-ff))))

> dxy-ff
#(extended-ff d 6 #(extended-ff x 7 #(extended-ff y 8 #(empty-ff))))
```


Procedimiento de Transformación

- Para convertir el conjunto de procedimientos que representan a un ADT:
  - Indentificar las expresiones lambda cuya evaluación produce objetos del ADT (i.e., función-finita)
  - Identicar las variable libres de estas expresiones lambda y asignar un campo en el registro correspondiente para cada una de éstas
  - Definir un procedimiento *apply* específico para el tipo de dato.

Extensión de un ADT

- Es posible extender nuevos procedimientos a un ADT sin modifar la representación interna; en este caso sólo hay que modificar la interface

  por ejemplo....

extend-ff*

- considerar el procedimiento extend-ff* que toma:
  - una lista de símbols *sym-list*
  - una lista de valores *val-list*
  - una función finita *ff*
- regresa:
  - una nueva función finita asociando *sym-list* con *val-list*
  - preserva todas las asociaciones de *ff* para símbolos que no estén en *sym-list*

extend-ff*

```
> (define extend-ff*
        (lambda (sym-list val-list ff)
                (if (null? sym-list)
                    ff
                    (extend-ff (car sym-list) (car val-list)
                            (extend-ff* (cdr sym-list) (cdr val-list) ff)))))
```


ribassoc: una utilería

- (ribassoc s los v fail-value)
  - *s* es un símbolo
  - *los* es una lista de símbolos
  - *v* es un vector
  - fail-value es un identificador de falla (e.g., un símbolo)
- tal que ribassoc regresa el valor en *v* que está asociado al símbolo *s* en *los*, o *fail-value* si esta asociación no está definida. Si la primera ocurrencia de *s* en *los* tiene índice *n*, el valor asociado a *s* es el enésimo valor en *v*.
- eg.

  (ribassoc 'b '(a b c) '#(1 2 3) 'fail) ⇒ 2

  (ribassoc 'd '(a b c) '#(1 2 3) 'fail) ⇒ fail

Agregar extend-ff* a la interface del ADT

- Representación procedural de extend-ff*

```
> (define extend-ff*
    (lambda (sym-list val-list ff)
        (lambda (symbol)
            (let ((val (ribassoc symbol
                                 sym-list
                                 (list->vector val-list)
                                 '*fail*)))
                (if (eq? val '*fail*)
                    (apply-ff ff symbol)
                    val)))))
```


Agregar extend-ff* a la interface del ADT

- Simplificando extend-ff*

```
> (define extend-ff*
    (lambda (sym-list val-list ff)
        (let ((val-vector (list->vector val-list)))
            (lambda (symbol)
                (let ((val (ribassoc symbol sym-list val-vector '*fail*)))
                    (if (eq? val '*fail*)
                        (apply-ff ff symbol)
                        val))))))
```


Representación de registros (Record Representation)

- Ya teníamos:

```
> (define-record extended-ff (sym val ff))
> (define extend-ff
      (lambda (sym val ff) (make-extended-ff sym val ff)))
```

- Ahora agregamos:

```
> (define-record extended-ff* (sym-list val-vector ff))
> (define-extend-ff*
      (lambda (sym-list val-list ff)
          (make-extended-ff* sym-list (list->vector val-list) ff)))
```


Nuevo apply-ff

```
> (define apply-ff
    (lambda (ff symbol)
        (variant-case ff
            (empty-ff ( ) (error  "empty-ff: no hay simbolo" symbol))
            (extended-ff (sym val ff)
                         (if (eq? symbol sym)
                             val
                             (apply-ff ff symbol)))
            (extended-ff* (sym-list val-vector ff)
                (let ((val (ribassoc symbol sym-list val-vector '*fail*)))
                    (if (eq? val '*fail*
                        (apply-ff ff symbol)
                        val)))
            (else (error  "apply-ff: función finita inválida" ff)))))
```


Ejemplos de extended-ff*

```
> (define ff1 (extend-ff*  '(d x y)  '(6  7  8) (create-empty-ff)))
> (define ff2 (extend-ff*  '(a b c)  '(1  2  3) ff1))
> (apply-ff ff2 'd)
6
> ff2
#(extended-ff* (a b c) #(1  2  3)
               #(extended-ff* (d x y) #(6  7  8) #(empty-ff)))
> (define-ff ff3 (extend-ff* '(d  e) '(4  5) ff2))
> (apply-ff ff3 'd)
4
> (apply-ff ff3  'a)
1
> ff3
#(extended-ff* (d  e) #(4 5)
    #(extended-ff* (a b c) #(1  2  3)
                   #(extended-ff* (d x y) #(6  7  8) #(empty-ff)))
```


Abstracción Sintáctica y de Datos (2)

- Abstracción de datos
- De las representaciones procedurales a las representaciones estructuradas (Data Structure Representations o DSR)
  - Representaciones procedurales
  - Representación de registros (Record Representation)
  - Representaciones alternativas de DSR

Representaciones alternativas de DSR

- Forma de *extended-ff** (sin *extended-ff*)

  #(extended-ff* $sym\text{-}list_n$ $val\text{-}vector_n$

  ...

  #(extended-ff* $sym\text{-}list_1$ $val\text{-}vector_1$

  #(empty-ff)) ...)

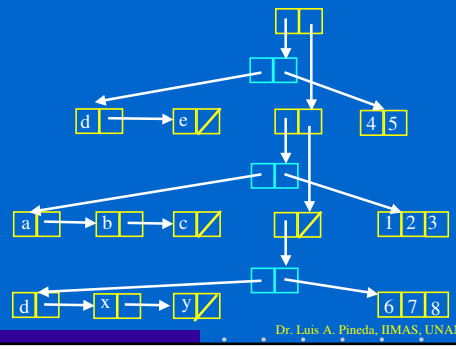
- Representación alternativa

  (($sym\text{-}list_n$. $val\text{-}vector_n$)

  ...

  ($sym\text{-}list_1$. $val\text{-}vector_1$))

estructura de “jaula de costillas” (ribcage environment structure)



FIN