

# Fundamentos de lenguajes de programación

## Abstracción de datos

Facultad de Ingeniería. Universidad del Valle

Junio de 2019

# Contenido

- 1 Introducción
- 2 Estrategias para representar tipos de datos
- 3 Definición de tipos de datos
- 4 Sintaxis Abstracta

# Abstracción de datos

- Cada vez que se define un conjunto de valores, se está definiendo un nuevo tipo de dato
- En un tipo de dato se define los valores, representaciones y operaciones sobre el mismo
- La representación de los datos es bastante compleja
- Para cambiar la representación de los datos se lleva a cabo una tarea llamada **Abstracción de datos**

# Abstracción de datos

- La abstracción de datos divide en dos los tipos de datos:  
**interface** e **implementación**
- La interfaz de un tipo de datos dice que representa el tipo de dato, sus operaciones y las propiedades de dichas operaciones
- La implementación proporciona una representación específica y un código para las operaciones que hacen uso de esa representación.

# Abstracción de datos

- Un tipo de dato que es abstracto se denomina Tipo Abstracto de Dato (TAD).
- El resto del programa fuera del tipo de dato, llamado el cliente del tipo de dato, manipula el nuevo dato solo mediante las operaciones especificadas en la interfaz
- El código del cliente es independiente de la representación si manipula los valores del tipo de dato solo a través de los procedimientos en su interfaz.

# Abstracción de datos

- Todo lo relacionado sobre el dato representado debe estar en el código de la implementación
- La parte más importante de la implementación es la especificación de como los datos son representados
- Se utiliza la implementación  $\lceil v \rceil$  para la representación de dato  $v$

Representación de números naturales:

$$\left\{ \begin{array}{l} (\text{zero}) = [0] \\ (\text{is-zero? } [n]) = \begin{cases} \text{\#t} & n = 0 \\ \text{\#f} & n \neq 0 \end{cases} \\ (\text{successor } [n]) = [n + 1] \quad (n \geq 0) \\ (\text{predecessor } [n + 1]) = [n] \quad (n \geq 0) \end{array} \right.$$

## Representación de números naturales:

- En la anterior especificación no se indica cómo se representan los números naturales
- A partir de la especificación se pueden escribir programas para la manipulación de los datos, sin importar su representación



Representación de números naturales:

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
        y
        (successor (plus (predecessor x) y))
    )
  )
)
```

Satisface  $(\text{plus } \lceil x \rceil \lceil y \rceil) = \lceil x + y \rceil$

# Abstracción de datos

## Ejemplo

Tres posibles representaciones para los enteros no negativos:

- 1 *Representación Unaria*: Donde un entero no negativo  $n$  es representado por una lista de  $n$  símbolos '#t'.

$$[0] = ()$$

$$[n + 1] = (\text{cons } \text{\#t } [n])$$

En esta representación, se satisface la especificación escribiendo:

```
(define zero '())  
(define iszero? null?)  
(define succ (lambda (n) (cons #t n)))  
(define pred cdr)
```

# Abstracción de datos

## Ejemplo

- 2 *Representación de Números de Scheme*: Se usa la representación interna de números de Scheme.

$$[n] = n$$

Se definen las entidades como:

```
(define zero 0)
(define iszero? zero?)
(define succ (lambda (n) (+ n 1)))
(define pred (lambda (n) (- n 1)))
```

# Abstracción de datos

## Ejemplo

- 3 *Representación Bignum*: Los números son representados en base  $N$ , para algún entero grande  $N$ . Dicha representación es una lista que consiste de números entre 0 y  $N - 1$ .

$$[n] = \begin{cases} () & n = 0 \\ (\text{cons } r [q]) & n = qN + r, 0 \leq r < N \end{cases}$$

Luego si  $N = 16$ , entonces:

$$\begin{aligned} [33] &= (1 \ 2) && ((1 \times 16^0) + (2 \times 16^1)) \\ [258] &= (\underline{2} \ 0 \ 1) && ((2 \times 16^0) + (0 \times 16^1) + (1 \times 16^2)). \end{aligned}$$

$$N=16$$

$$(1 \ 2 \ 3)$$

$$(2 \ 2 \ 3)$$

⋮

$$\begin{aligned} & (15 \ 2 \ 3) \\ \hookrightarrow & (0 \ 3 \ 3) \end{aligned}$$

$$1 \times 16^0 + 2 \times 16^1 + 3 \times 16^2$$

$$\begin{array}{c} 909 \\ \downarrow + \\ 910 \end{array}$$

$$\begin{array}{ccccccc} (15 & 15 & 15 & 15 & 15 & 15 & 15 \ 2) \\ \hline (0 & 0 & 0 & 0 & 0 & 0 & 0 \ 3) \end{array}$$

$$(15 \ 15 \ 3)$$

$$(0 \ (\text{Succ } (15 \ 3)))$$

$$(C)$$

$$\begin{array}{c} 0 \quad 0 \quad 4 \\ (15 \ 15 \ 15 \ 15) \end{array}$$

$$\text{cons } 0 \ (\text{Succ } (C \ 15 \ 15 \ 15))$$

$$\text{cons } 0 \ (\text{cons } 0 \ (\text{Succ } (15 \ 15)))$$

$$\text{cons } 0 \ (\text{cons } 0 \ (\text{cons } 0 \ (\text{Succ } (15))))$$

$$\text{cons } 0 \ (\text{cons } 0 \ (\text{cons } 0 \ (\text{cons } 0 \ (\text{Succ } (C))))))$$

$$\text{cons } 0 \ (\text{cons } 0 \ (\text{cons } 0 \ (\text{cons } 0 \ (\text{Succ } (1))))))$$

$$(3 \ 2 \ 1)$$

$$(2 \ 2 \ 1)$$

$$(0 \ 3 \ 2)$$

↓

$$(15 \ 2 \ 2)$$

$$(0 \ 0 \ 0 \ 1)$$

↓

$$(15 \ 15 \ 15)$$

$$(0 \ 0 \ 0 \ 4 \ 2)$$

↓

$$(15 \ 15 \ 15 \ 3 \ 2)$$

# Contenido

- 1 Introducción
- 2 Estrategias para representar tipos de datos
- 3 Definición de tipos de datos
- 4 Sintaxis Abstracta

# Estrategias para representar tipos de datos

- Cuando es usada la abstracción de datos, los programas tienen la propiedad de independencia de representación.
- Se presentan algunas estrategias para representar tipos de datos. Se ilustran estas estrategias usando el tipo de dato *ambiente*.
- Un ambiente asocia un valor con cada elemento de un conjunto finito de variables.
- Un ambiente puede ser usado para asociar las variables con sus valores en la implementación de un lenguaje de programación.



# Estrategias para representar tipos de datos

- Las variables pueden ser representadas de cualquier manera, siempre y cuando sea posible chequear la igualdad entre dos variables.
- Las variables se pueden representar mediante símbolos, cadenas , referencias en una tabla hash o incluso mediante números.

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Interfaz

$X \longrightarrow ?$

- Un ambiente es una función cuyo dominio es un conjunto finito de variables y cuyo rango es el conjunto de todos los valores de Scheme.
- De acuerdo a la notación matemática, los ambientes representan todos los conjuntos de la forma  $\{(s_1, v_1), \dots, (s_n, v_n)\}$ , donde los  $s_i$  son símbolos diferentes y los  $v_i$  son valores de Scheme.

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Interfaz

La interfaz del tipo de dato ambiente tiene tres procedimientos:

<code>(empty-env)</code>	$= [\emptyset]$
<code>(apply-env <math>[f]</math> <math>s</math>)</code>	$= f(s)$
<code>(extend-env <math>var</math> <math>v</math> <math>[f]</math>)</code>	$= [g],$

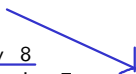
donde  $g(s') = \begin{cases} v & \text{si } s' = var \\ f(s') & \text{de otra forma} \end{cases}$

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Interfaz

- El procedimiento `empty-env` debe producir una representación del ambiente vacío.
- El procedimiento `apply-env` aplica una representación de un ambiente a un argumento.
- El procedimiento `(extend-env var val env)` produce un nuevo ambiente que se comporta como `env`, excepto que su valor en el símbolo `var` es `val`.

```
(define e
  (extend-env 'd 6
    (extend-env 'y 8
      (extend-env 'x 7
        (extend-env 'y 14
          (empty-env)))))))
```



$(\text{apply-env } e \text{ 'd}) \rightarrow 6$   
 $(\text{apply-env } e \text{ 'y}) \rightarrow 8$

### Representación como estructura de datos:

- Cada ambiente puede ser construido mediante una expresión en la siguiente gramática:

```
<env-exp> ::= (empty-env)
           ::= (extend-env <identificador>
                          <scheme-value> <env-exp>)
```

- De acuerdo a esta gramática los ambientes pueden ser representados como listas en Scheme.

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Implementación

## Representación como estructura de datos:

**empty-env:**  $() \rightarrow Env$

```
(define empty-env  
  (lambda () (list 'empty-env)))
```

**extend-env:**  $Var \times SchemeValue \times Env \rightarrow Env$

```
(define extend-env  
  (lambda (var val env)  
    (list 'extend-env var val env)))
```

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Implementación

## Representación como estructura de datos:

**apply-env:**  $Env \times Var \rightarrow SchemeValue$

```
(define apply-env
  (lambda (env search-var)
    (cond ((eqv? (car env) 'empty-env)
           (report-no-binding-found search-var))
          ((eqv? (car env) 'extend-env)
           (let ((saved-var (cadr env))
                 (saved-val (caddr env))
                 (saved-env (cadddr env)))
             (if (eqv? search-var saved-var)
                 saved-val
                 (apply-env saved-env search-var))))
          (else (report-invalid-env env))))))
```

### Representación Procedimental:

- La interfaz del tipo de dato ambiente tiene una propiedad importante: ella tiene exactamente una entidad *observadora* *apply-env*.
- Esto permite representar un ambiente como un procedimiento que toma una variable y retorna su valor asociado.
- Se define *empty-env* y *extend-env* de tal manera que retornan un procedimiento que al ser aplicado devuelve el valor asociado a la variable en el ambiente.



# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Implementación

## Representación Procedimental:

**empty-env:**  $() \rightarrow Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

**extend-env:**  $Var \times SchemeValue \times Env \rightarrow Env$

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```



Universidad  
del Valle

# Estrategias para representar tipos de datos

Tipo de dato Ambiente: Implementación

## Representación Procedimental:

**apply-env:**  $Env \times Var \rightarrow SchemeValue$

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

# Contenido

- 1 Introducción
- 2 Estrategias para representar tipos de datos
- 3 Definición de tipos de datos**
- 4 Sintaxis Abstracta

# Interfaces para tipos de datos recursivos

Hasta el momento hemos visto como definir tipos de datos recursivos mediante diferentes métodos. Por ejemplo:

```
<Lc-exp> ::= <var-exp>      (<identifier>)  
          ::= <lambda-exp>   (lambda (<identifier>) <Lc-exp>)  
          ::= <app-exp>       (<Lc-exp> <Lc-exp>)
```

# Interfaces para tipos de datos recursivos

Por ejemplo para una representación basada en listas:

```
;;x  
'x  
;;(lambda (x) x)  
(list 'lambda x 'x)  
;;( (lambda (x) (lambda (y) (x y))) x)  
(list 'lambda 'x (list 'lambda 'y (list 'x 'y)))
```

# Interfaces para tipos de datos recursivos

Así mismo, hemos intentado manipular y operar tipos de datos recursivos. Por ejemplo, para las expresiones del cálculo  $\lambda$  se han definido procedimientos como `occurs-free?`:

```
(define occurs-free?
  (lambda (var exp)
    (cond
      ;; Caso var-exp
      [(symbol? exp) (eqv? var exp)]
      ;; Caso lambda-exp
      [(eqv? (car exp) 'lambda)
       (and (not (eqv? var (car (cadr exp))))
            (occurs-free? var (caddr exp)))]
      ;; Caso app-exp
      [else
       (or (occurs-free? var (car exp))
           (occurs-free? var (cadr exp)))]))
```

# Interfaces para tipos de datos recursivos

- No obstante, esta definición de `occurs-free?` es difícil de leer (es difícil decir que `(car (cadr exp))` se refiere a la declaración de una variable en una expresión lambda o que `(caddr exp)` se refiere a su cuerpo).
- Esta definición establece cierta dependencia con la implementación de las expresiones del cálculo  $\lambda$  como listas.

# Interfaces para tipos de datos recursivos

- Se puede mejorar estos aspectos introduciendo interfaces para la creación y manipulación de datos de un cierto tipo.
- Una interfaz para un tipo de dato consta de procedimientos *constructores* y procedimientos *observadores*.
- Los procedimientos observadores pueden ser predicados o extractores.



# Interfaces para tipos de datos recursivos

## Interfaz para expresiones cálculo $\lambda$

Para las expresiones del cálculo  $\lambda$  tenemos la siguiente interfaz:

### ■ Constructores:

`var-exp` :  $Var \rightarrow Lc-exp$

`lambda-exp` :  $Var \times Lc-exp \rightarrow Lc-exp$

`app-exp` :  $Lc-exp \times Lc-exp \rightarrow Lc-exp$

### ■ Predicados:

`var-exp?` :  $Lc-exp \rightarrow Bool$

`lambda-exp?` :  $Lc-exp \rightarrow Bool$

`app-exp?` :  $Lc-exp \rightarrow Bool$

# Interfaces para tipos de datos recursivos

Interfaz para expresiones cálculo  $\lambda$

## ■ Extractores:

<code>var-exp-&gt;var</code>	$: Lc-exp \rightarrow Var$
<code>lambda-exp-&gt;bound-var</code>	$: Lc-exp \rightarrow Var$
<code>lambda-exp-&gt;body</code>	$: Lc-exp \rightarrow Lc-exp$
<code>app-exp-&gt;rator</code>	$: Lc-exp \rightarrow Lc-exp$
<code>app-exp-&gt;rand</code>	$: Lc-exp \rightarrow Lc-exp$

# Interfaces para tipos de datos recursivos

## Interfaz para expresiones cálculo $\lambda$

Ahora es posible escribir una versión de `occurs-free?` que depende solo de la interfaz.

```
(define occurs-free?
  (lambda (search-var exp)
    (cond
      ((var-exp? exp) (eqv? search-var (var-exp->var exp)))
      ((lambda-exp? exp)
       (and
        (not (eqv? search-var (lambda-exp->bound-var exp)))
        (occurs-free? search-var (lambda-exp->body exp))))
      (else
       (or
        (occurs-free? search-var (app-exp->rator exp))
        (occurs-free? search-var (app-exp->rand exp)))))))
```

# Interfaces para tipos de datos recursivos

- La interfaz para el tipo de dato ambiente consta de los constructores `empty-env` y `extended-env` y del procedimiento observador `apply-env`.
- La interfaz para el tipo de dato lista consta de los constructores `empty-list` y `cons`, de los procedimientos extractores `car` y `cdr` y del predicado `list?`.

# Interfaces para tipos de datos recursivos

## Diseño de interfaces para tipos de datos recursivos

### Receta general para el diseño de interfaces de datos recursivos

- 1 Incluir un constructor para cada clase de dato (regla de producción) en el tipo de dato.
- 2 Incluir un predicado para cada clase de dato en el tipo de dato.
- 3 Incluir un extractor para cada pieza de dato pasada a un constructor del tipo de dato.

<arbol - ternario>

```
 ::= (arbol-vacio) '()
```

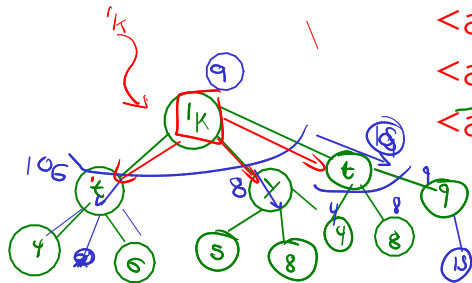
```
::= (arbol-hoja) <numero>
```

```
::= (arbol-nodo) <simbolo>
```

<arbol - ternario>

<arbol - ternario>

<arbol-ternario>



# Interfaces para tipos de datos recursivos

## Interfaz `define-datatype`

- Para tipos de datos complejos, resulta tedioso construir interfaces rápidamente.
- La interfaz `define-datatype` es una herramienta de Scheme para construir e implementar interfaces para tipos de datos.

# Interfaces para tipos de datos recursivos

## Interfaz define-datatype

- Las interfaces son especificadas mediante la expresión `define-datatype` que tiene la forma general:

```
(define-datatype nombre-tipo nombre-predicado-tipo  
  {(nombre-variante {(nombre-campo predicado )}* )}* )
```

- Esta declaración crea un tipo de dato llamado *nombre-tipo* con algunas variantes.
- Cada variante tiene un nombre (*nombre-variante*) y cero o más campos, cada uno con un nombre y un predicado asociado.



# Interfaces para tipos de datos recursivos

## Interfaz `define-datatype`

- Dos tipos no pueden tener el mismo nombre, tampoco dos variantes, aunque pertenezcan a tipos diferentes, pueden tener el mismo nombre.
- Para cada variante, un procedimiento constructor es creado. Si hay  $n$  campos en una variante, su constructor recibe  $n$  argumentos, prueba sí cada uno de ellos satisface el predicado asociado y retorna un nuevo valor de dicha variante del tipo de dato.
- El nombre *nombre-predicado-tipo* es ligado a un predicado. Este predicado determina sí su argumento es un valor perteneciente al tipo `nombre-tipo`.

# Interfaces para tipos de datos recursivos

## Interfaz define-datatype: Ejemplos

### Expresiones cálculo $\lambda$

```
(define-datatype lc-exp lc-exp?  
  (var-exp (id identifier?))  
  (lambda-exp (id identifier?)  
               (body lc-exp?))  
  (app-exp (rator lc-exp?)  
            (rand lc-exp?)))
```

# Interfaces para tipos de datos recursivos

## Interfaz define-datatype: Ejemplos

### Tipo de dato s-list

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list (first s-exp?)
                    (rest s-list?)))

(define-datatype s-exp s-exp?
  (symbol-s-exp (sym symbol?))
  (s-list-s-exp (slst s-list?)))
```

# Interfaces para tipos de datos recursivos

## Interfaz define-datatype: Ejemplos

### Tipo de dato s-list

```
(define-datatype s-list s-list?
  (an-s-list (sexps (list-of s-exp?))))

(define list-of
  (lambda (pred)
    (lambda (val)
      (or (null? val)
          (and (pair? val)
                (pred (car val))
                ((list-of pred) (cdr val))))))))
```

# Interfaces para tipos de datos recursivos

## cases

- Para determinar a que objeto de un tipo de dato pertenece una variante y extraer sus componentes, se usa la forma `cases`, la cual tiene la sintaxis general:

```
(cases nombre-tipo expresion  
  {(nombre-variante ( {nombre-campo}*) consecuente)}*  
  (else por-defecto))
```

# Interfaces para tipos de datos recursivos

## cases

- La expresión `cases` evalúa *expresion*. Esto da como resultado un valor *v* de tipo *nombre-tipo*.
- Si *v* es una variante *nombre-variante*, cada uno de los campos *nombre-campo* son asociados al valor del correspondiente campo de *v*. Luego la expresión *consecuente* es evaluada y su valor es retornado.
- Si *v* no es una de las variantes y la cláusula `else` es especificada, la expresión *por-defecto* es evaluada y su valor retornado.
- Si no existe una cláusula `else`, entonces tiene que existir una variante para todos los tipos de dato.

# Interfaces para tipos de datos recursivos

cases: Ejemplo

```
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp (var)
        (eqv? var search-var))
      (lambda-exp (bound-var body)
        (and (not (eqv? search-var bound-var))
              (occurs-free? search-var body)))
      (app-exp (rator rand)
        (or (occurs-free? search-var rator)
            (occurs-free? search-var rand))))))
```

# Interfaces para tipos de datos recursivos

Más ejemplos

## Tipo de dato bin-tree

```
(define-datatype bintree bintree?
  (leaf-node (datum number?))
  (interior-node (key symbol?)
                 (left bintree?)
                 (right bintree?)))
```



# Interfaces para tipos de datos recursivos

## Más ejemplos

### Tipo de dato bin-tree

Definir un procedimiento que permita encontrar la suma de los enteros en las hojas de un árbol. Usando cases se tiene:

```
(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node (datum) datum)
      (interior-node (key left right)
        (+ (leaf-sum left) (leaf-sum right))))))
```

# Interfaces para tipos de datos recursivos

## Más ejemplos

### Ambientes

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record (syms (list-of symbol?))
                        (vals (list-of scheme-value?))
                        (env environment?)))

(define scheme-value? (lambda (v) #t))
```

# Interfaces para tipos de datos recursivos

## Más ejemplos

### Ambientes

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env
          "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (number? pos)
              (list-ref vals pos)
              (apply-env env sym)))))))
```

# Contenido

- 1 Introducción
- 2 Estrategias para representar tipos de datos
- 3 Definición de tipos de datos
- 4 Sintaxis Abstracta

- Dada la gramática de las expresiones del cálculo lambda:

$$\begin{aligned}\langle \text{expresión} \rangle &::= \langle \text{identificador} \rangle \\ &::= (\text{lambda } (\langle \text{identificador} \rangle) \langle \text{expresión} \rangle) \\ &::= (\langle \text{expresión} \rangle \langle \text{expresión} \rangle)\end{aligned}$$

- Se puede representar cada expresión del cálculo lambda usando el tipo de dato `lc-exp` definido anteriormente mediante `define-datatype` .

# Sintaxis Abstracta

`for(int i=0; ...)`

- Una BNF especifica una representación particular de un tipo de dato que usa los valores generados por la gramática
- Esta representación es llamada sintaxis concreta o representación externa **Afuera (Programador**
- Para procesar dichos datos, se requiere convertirlos a una representación interna o sintaxis abstracta, en la cual los símbolos terminales (como paréntesis) no necesiten ser almacenados ya que no llevan información

# Sintaxis Abstracta

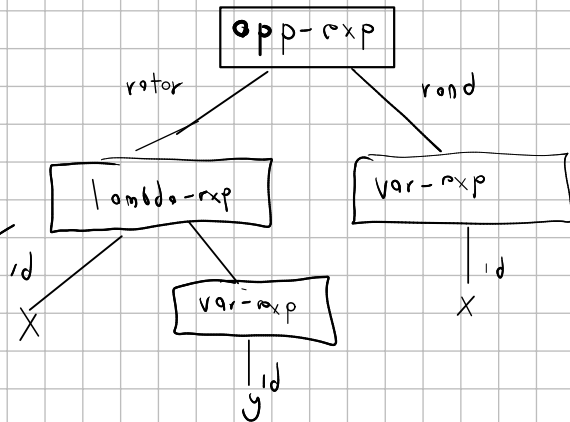
- Para crear una sintaxis abstracta a partir de una sintaxis concreta, se debe nombrar cada regla de producción de la sintaxis concreta y cada ocurrencia de un símbolo no terminal
- Para la gramática de las expresiones del cálculo  $\lambda$ , se puede resumir las opciones (sintaxis concreta y abstracta) usando la siguiente notación:

$\langle \text{expresión} \rangle ::= \langle \text{identificador} \rangle$   
 $\text{var-exp } (\text{id})$

$::= (\text{lambda } (\langle \text{identificador} \rangle) \langle \text{expresión} \rangle)$   
 $\text{lambda-exp } (\text{id } \text{body})$

$::= (\langle \text{expresión} \rangle \langle \text{expresión} \rangle)$   
 $\text{app-exp } (\text{rator rand})$

(app-exp (lambda-exp 'x  
 (~~(var-exp 'y)~~) )  
 (var-exp 'x))



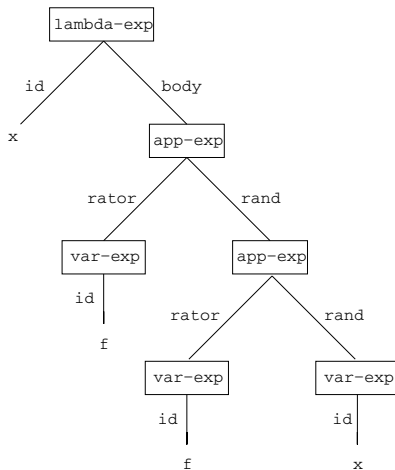
```

> (app-exp (lambda-exp 'x ((var-exp 'y))) (var-exp 'x))
#(struct:app-exp #(struct:lambda-exp x #(struct:var-exp y)) #(struct:var-exp x))
  
```



- La sintaxis abstracta de una expresión es más fácil de comprender visualizándola como un *árbol de sintaxis abstracta*
- El siguiente ejemplo muestra el árbol para la expresión `(lambda (x) (f (f x)))`:

# Sintaxis Abstracta



- Los árboles de sintaxis son útiles en lenguajes de programación de procesamiento de sistemas ya que los programas que procesan otros programas (como los interpretadores o compiladores) son casi siempre *dirigidos por sintaxis*
- Esto es que cada parte de un programa es guiado por la regla gramatical asociada con dicha parte, y cualquier subparte correspondiente a un símbolo no terminal puede ser accedido con facilidad

- Cuando se procesa la expresión  $(\text{lambda } (x) (f (f x)))$ , primero se debe reconocer como una expresión del cálculo lambda, correspondiente a la regla:

$$\langle \text{expresión} \rangle ::= (\text{lambda } (\langle \text{identificador} \rangle) \langle \text{expresión} \rangle)$$

- El parámetro formal es  $x$  y el cuerpo es  $(f (f x))$ . El cuerpo debe ser reconocido como una app-exp, y así sucesivamente.

```

<arbol-ternario > ::= <int>
                    (arbol-hoja (num))
                    { ::= <simbolo> <arbol-ternario ><arbol-ternario >
                      <arbol-ternario >
                      (arbol-novacio (sym h1 h2 h3))
                    }

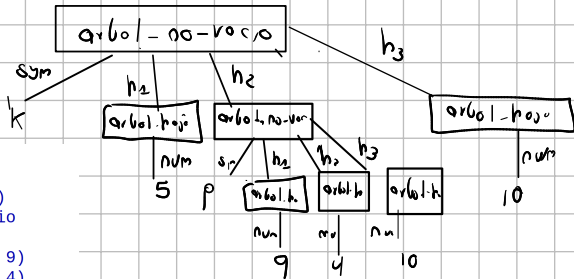
```

AREA DEL PROGRAMADOR

```

(define arbolMortal
  (arbol-no-vacio
    'k
    (arbol-hoja 5)
    (arbol-no-vacio 'p (arbol-hoja 9) (arbol-hoja 4) (arbol-hoja 10))
    (arbol-hoja 10)
  ))

```



```

#(struct:arbol-no-vacio
  k
  #(struct:arbol-hoja 5)
  #(struct:arbol-no-vacio
    p
    #(struct:arbol-hoja 9)
    #(struct:arbol-hoja 4)
    #(struct:arbol-hoja 10))
  #(struct:arbol-hoja 10))

```

$\langle \text{lista-s} \rangle ::= '()'$   
 $\quad (\text{listas-vacia})$

$\quad ::= \langle \text{simbolo} \rangle \langle \text{lista-s} \rangle$   
 $\quad (\text{lista-s-novacia})$

# Sintaxis Abstracta

El problema de convertir un árbol de sintaxis abstracta a una representación lista-y-símbolo (con lo cual Scheme mostraría las expresiones en su sintaxis concreta), se resuelve con el procedimiento:

```
(define unparsed-expression
  (lambda (exp)
    (cases expression exp
      (var-exp (id) id)
      (lambda-exp (id body)
        (list 'lambda (list id)
              (unparsed-expression body)))
      (app-exp (rator rand)
        (list (unparsed-expression rator)
              (unparsed-expression rand)))))
```

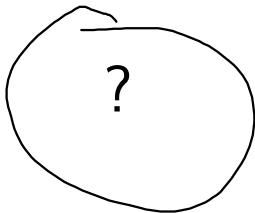
- La tarea de derivar el árbol de sintaxis abstracta a partir de una cadena de caracteres es denominado *parsing*, y es llevado a cabo por un programa llamado *parser* (analizador sintáctico)
- El siguiente procedimiento deriva la representación en sintaxis concreta a árboles de sintaxis abstracta:



# Sintaxis Abstracta

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
           (lambda-exp (caadr datum)
                        (parse-expression (caddr datum)))
           (app-exp
            (parse-expression (car datum))
            (parse-expression (cadr datum)))))
      (else (eopl:error 'parse-expression
                         "Invalid concrete syntax ~s" datum)))))
```

# Preguntas



# Próxima sesión

- Semántica de los conceptos fundamentales de los lenguajes de programación.
- Primer interpretador.