

*Home Page*

*Title Page*

*Contents*



*Page 1 of 100*

*Go Back*

*Full Screen*

*Close*

*Quit*

# La Abstracción de Datos: Técnica Fundamental en Programación

Profesor: Juan Francisco Diaz

[jdiaz@eisc.univalle.edu.co](mailto:jdiaz@eisc.univalle.edu.co)

Asistente de Docencia: Gerardo M. Sarria M.

[gsarria@eisc.univalle.edu.co](mailto:gsarria@eisc.univalle.edu.co)

April 10, 2002

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 2 of 100

Go Back

Full Screen

Close

Quit

## Concepto General

La técnica de *abstracción de datos* se usa para localizar todas las partes de un programa que son dependientes de la representación.

La abstracción de datos divide un tipo de dato en dos partes: una *interfaz* que dice qué representa el tipo de dato, cuáles son las operaciones de los datos y cuáles propiedades deben tener dichas operaciones; y una *implementación* que provee una representación específica de los datos y un código para las operaciones que hacen uso de esa representación.

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 sólo mediante las operaciones especificadas en la interfaz.

Home Page

Title Page

Contents



Page 3 of 100

Go Back

Full Screen

Close

Quit

## Concepto General (cont.)

Toda la representación de los datos debe estar en el código de la implementación. Se usará la notación  $\lceil v \rceil$  para “la representación del dato  $v$ ”.

Ejemplo 1:

El tipo de dato *entero no negativo*

La interfaz consiste de cuatro entidades: una constante **zero** y tres procedimientos, **iszero?**, **succ**, y **pred**.

El comportamiento de los procedimientos se especifica así:

**zero** =  $\lceil 0 \rceil$

$(\text{iszero? } \lceil n \rceil) = \begin{cases} \text{\#t} & n = 0 \\ \text{\#f} & n \neq 0 \end{cases}$

$(\text{succ } \lceil n \rceil) = \lceil n + 1 \rceil \quad (n \geq 0)$

$(\text{pred } \lceil n + 1 \rceil) = \lceil n \rceil \quad (n \geq 0)$

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 4 of 100

Go Back

Full Screen

Close

Quit

## Concepto General (cont.)

La especificación anterior no dice cómo se representarán los enteros no negativos, simplemente asegura que **zero** tendrá que estar limitado a la representación de 0, que el procedimiento **succ**, dada la representación de  $n$ , debe retornar la representación del entero  $n + 1$ , etc.

De allí (la especificación), se pueden escribir programas que manipulan los enteros no negativos, asegurando que serán correctos, no importa cuál sea la representación que se use.

Ejemplo 2:

```
(define plus
  (lambda (x y)
    (if (iszero? x)
        y
        (succ (plus (pred x) y)))))
```

satisfará  $(\text{plus } [x] [y]) = [x + y]$ , no importa la implementación de enteros no negativos que se use.

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 5 of 100

Go Back

Full Screen

Close

Quit

## Concepto General (cont.)

Tres posibles representaciones para los enteros no negativos pueden ser:

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

$$[0] = ()$$

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

En esta representación, se satisface la especificación escribiendo

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

## Concepto General (cont.)

2. *Representación de Números de Scheme*: Se usa la representación interna de números de Scheme.  $[n] = n$

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

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

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

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 7 of 100

Go Back

Full Screen

Close

Quit

## Interfaz: define-datatype

Las interfaces son especificadas mediante `define-datatype`. Esta declaración sólo puede aparecer al inicio de un programa y tiene la forma general

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

Esta declaración crea un tipo de dato **registro-variante** (**unión discriminada**<sup>a</sup> de tipos de registro), llamado *nombre-tipo*.

El *nombre-predicado-tipo* está relacionado con un **predicado**, el cual determina si su argumento es un valor que pertenece al tipo nombrado.

Cada **variante**<sup>b</sup> tiene un *nombre-variante* y cero o más **campos**<sup>c</sup>, cada uno con su *nombre-campo* y un *predicado* asociado.

---

<sup>a</sup>Un tipo **unión** es aquel cuyos valores son de alguno de los múltiples tipos dados. Los valores de un tipo de **unión discriminada** contienen un valor de los tipos de unión y una etiqueta indicando a cuál tipo pertenece el valor.

<sup>b</sup>Los tipos de datos definidos son representados como una unión discriminada de tipos de registros llamados **registros variantes**. Cada tipo de registro es llamado **variante** del tipo.

<sup>c</sup>Un tipo de dato **agregado** es aquel que contiene valores de otros tipos como arreglos y registros. En este contexto, un registro es denominado **campo**.

*Home Page*

*Title Page*

*Contents*



*Page 8 of 100*

*Go Back*

*Full Screen*

*Close*

*Quit*

## define-datatype (cont.)

Dos tipos no pueden tener el mismo nombre y dos variantes, aún si pertenecen a distintos tipos, no pueden tener el mismo nombre. Además, los nombres de tipos no pueden ser usados como nombres de variantes.

Para cada variante se crea un nuevo procedimiento llamado **constructor** el cual crea los valores de los datos que pertenecen a cada variante.



Home Page

Title Page

Contents



Page 9 of 100

Go Back

Full Screen

Close

Quit

## define-datatype (cont.)

Ejemplo 3:

Considérese la gramática de una  $\langle \text{lista-s} \rangle$ .

$$\begin{aligned}\langle \text{lista-s} \rangle &::= (\{\langle \text{expresión-símbolo} \rangle\}^*) \\ \langle \text{expresión-símbolo} \rangle &::= \langle \text{símbolo} \rangle \mid \langle \text{lista-s} \rangle\end{aligned}$$

Los datos en una  $\langle \text{lista-s} \rangle$  pueden ser representados por el tipo de datos `s-list` definido así:

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

(define-datatype symbol-exp symbol-exp?
  (symbol-symbol-exp
    (data symbol?))
  (s-list-symbol-exp
    (data s-list?)))
```

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 10 of 100

Go Back

Full Screen

Close

Quit

## define-datatype (cont.)

El tipo de dato `s-list` tiene su propia representación de las listas usando `(empty-s-list)` y `non-empty-s-list` en lugar de `()` y `cons`. Este tipo de dato, usando las listas de Scheme, se puede escribir así:

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

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

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 11 of 100

Go Back

Full Screen

Close

Quit

## Interfaz: cases

Para determinar a qué 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 expresión
  {(nombre-variante ({nombre-campo}*) consecuente)}*
  (else por-defecto) )
```

Lo que hace esta forma es evaluar *expresión*, cuyo resultado es un valor *v* de *nombre-tipo*. Si *v* es una variante de *nombre-variante*, cada uno de los *nombre-campos* son asociados al valor del correspondiente campo de *v*, 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 es retornado; si no existe una cláusula **else**, entonces tiene que existir una variante para todos los tipos de dato.

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 12 of 100

Go Back

Full Screen

Close

Quit

## cases (cont.)

Ejemplo 4:

Considérese el tipo de dato `bintree` definido así:

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

Y se desea encontrar la suma de los enteros en las hojas de un árbol. Este problema se resuelve usando `cases` y escribiendo:

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

Home Page

Title Page

Contents



Page 13 of 100

Go Back

Full Screen

Close

Quit

# 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 definido como

```
(define-datatype expression expression?
  (var-exp
    (id symbol?))
  (lambda-exp
    (id symbol?)
    (body expression?))
  (app-exp
    (rator expression?)
    (rand expression?)))
```

Home Page

Title Page

Contents



Page 14 of 100

Go Back

Full Screen

Close

Quit

## Sintaxis Abstracta (cont.)

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*.

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.

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 15 of 100

Go Back

Full Screen

Close

Quit

## Sintaxis Abstracta (cont.)

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.

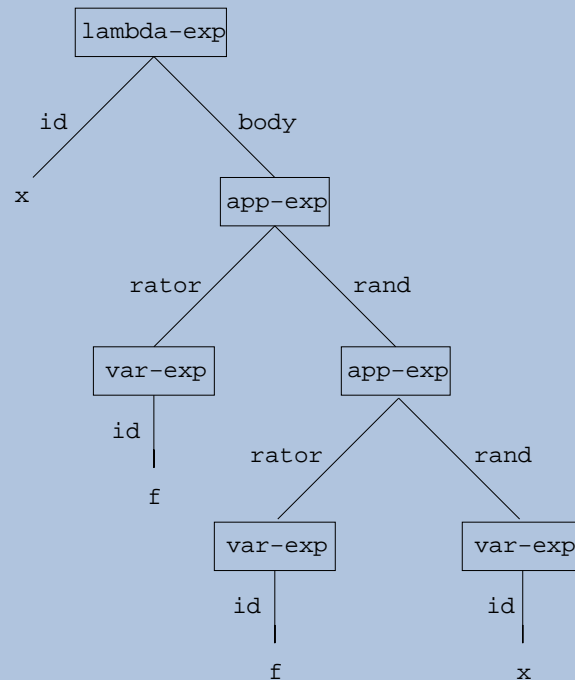
Para la gramática de las expresiones del cálculo lambda, se puede resumir las opciones (sintaxis concreta y abstracta) usando la siguiente notación:

$$\begin{aligned}\langle \text{expresión} \rangle &::= \langle \text{identificador} \rangle \\ &\quad \boxed{\text{var-exp (id)}} \\ &::= (\text{lambda } (\langle \text{identificador} \rangle) \langle \text{expresión} \rangle) \\ &\quad \boxed{\text{lambda-exp (id body)}} \\ &::= (\langle \text{expresión} \rangle \langle \text{expresión} \rangle) \\ &\quad \boxed{\text{app-exp (rator rand)}}\end{aligned}$$

## Sintaxis Abstracta (cont.)

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)))`.

Ejemplo 5:





Home Page

Title Page

Contents

◀

▶

◀

▶

Page 17 of 100

Go Back

Full Screen

Close

Quit

## Sintaxis Abstracta (cont.)

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.

Ejemplo 6:

Cuando se procesa la expresión `(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.

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 18 of 100

Go Back

Full Screen

Close

Quit

## Sintaxis Abstracta (cont.)

Ejemplo 7:

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

## Sintaxis Abstracta (cont.)

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).

La rutina `read` de Scheme deriva automáticamente cadenas a listas y símbolos. El siguiente procedimiento deriva esas estructuras de listas a árboles de 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)))))
```

*Home Page*

*Title Page*

*Contents*



Page *20* of *100*

*Go Back*

*Full Screen*

*Close*

*Quit*

## Implementación

Cuando una abstracción de datos es usada, los programas tienen la propiedad de ser independientes de la representación particular usada para implementar ese tipo abstracto de datos.

Por lo anterior es posible cambiar la representación redefiniendo un pequeño número de procedimientos que pertenecen a la interfaz.

## Implementación: Ambientes

Un ambiente es una función cuyo dominio es un conjunto finito de símbolos de Scheme, y cuyo rango es el conjunto de todos los valores de Scheme.

Matemáticamente los conjuntos se representan como  $\{(s_1, v_1), \dots, (s_n, v_n)\}$ , donde los  $s_i$  son símbolos diferentes y los  $v_i$  son valores de Scheme.

La interfaz de este tipo de dato tiene tres procedimientos:

`(empty-env)` =  $\lceil \emptyset \rceil$

`(apply-env  $\lceil f \rceil$  s)` =  $f(s)$

`(extend-env`

$\lceil s_1 \dots s_k \rceil$

$\lceil v_1 \dots v_k \rceil$

$\lceil f \rceil$ )

=  $\lceil g \rceil$ ,

donde  $g(s') = \begin{cases} v_i & \text{si } s' = s_i \text{ para algún } i, 1 \leq i \leq k \\ f(s') & \text{de otra forma} \end{cases}$

El procedimiento `empty-env` debe producir una representación del ambiente vacío; `apply-env` aplica una representación de un ambiente a un argumento; y `(extend-env  $\lceil s_1 \dots s_n \rceil \lceil v_1 \dots v_n \rceil env)$`  produce un nuevo ambiente que se comporta como  $env$ , excepto que su valor en el símbolo  $s_i$  es  $v_i$ .

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 22 of 100

Go Back

Full Screen

Close

Quit

## Representación de Ambientes: Procedimental

Un objeto de *primera-clase* es aquél que puede ser pasado como argumento, retornado como valor, y almacenado en una estructura de datos. Los procedimientos de Scheme son de primera-clase.

Un ambiente puede ser representado como un procedimiento de Scheme que toma un símbolo y retorna su valor asociado. Con esta representación, la interfaz de un ambiente puede ser definida como

```
(define empty-env
  (lambda ()
    (lambda (sym)
      (eopl:error 'apply-env "No binding for ~s" sym))))

(define extend-env
  (lambda (syms vals env)
    (lambda (sym)
      (let ((pos (list-find-position sym syms)))
        (if (number? pos)
            (list-ref vals pos)
            (apply-env env sym)))))))
```

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 23 of 100

Go Back

Full Screen

Close

Quit

## Representación Procedimental (cont.)

```
(define apply-env
  (lambda (env sym)
    (env sym)))

(define list-find-position
  (lambda (sym los)
    (list-index (lambda (sym1) (eqv? sym1 sym)) los)))

(define list-index
  (lambda (pred ls)
    (cond
      ((null? ls) #f)
      ((pred (car ls)) 0)
      (else (let ((list-index-r (list-index pred (cdr ls))))
                (if (number? list-index-r)
                    (+ list-index-r 1)
                    #f)))))))
```

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 24 of 100

Go Back

Full Screen

Close

Quit

## Representación de Ambientes: Árbol de Sintaxis Abstracta

Cada ambiente es construido empezando con el ambiente vacío y aplicando `extend-env`  $n$  veces, para  $n \geq 0$ .

Las expresiones con las que puede ser construido un ambiente se describen con la siguiente gramática

```
⟨env-rep⟩ ::= (empty-env)  
           empty-env-record  
  
          ::= (extend-env ({⟨símbolo⟩}*) ({⟨valor⟩}*) ⟨env-rep⟩)  
           extended-env-record (syms vals env)
```



Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 25 of 100

Go Back

Full Screen

Close

Quit

## Representación de Árbol de Sintaxis Abstracta (cont.)

El árbol de sintaxis abstracta para esta gramática puede ser definido como

```
(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))
```

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 26 of 100

Go Back

Full Screen

Close

Quit

## Representación de Árbol de Sintaxis Abstracta (cont.)

Se puede implementar la abstracción del ambiente redefiniendo los procedimientos `empty-env` y `extend-env` para construir las variantes apropiadas, y redefiniendo `apply-env` para interpretar la información en los registros y realizar las acciones especificadas en el cuerpo de la expresión `(lambda (sym) ...)`.

La implementación del tipo de dato usando la nueva representación es:

```
(define empty-env
  (lambda ()
    (empty-env-record)))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms vals env)))
```

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 27 of 100

Go Back

Full Screen

Close

Quit

## Representación de Árbol de Sintaxis Abstracta (cont.)

```
(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)))))))
```

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 28 of 100

Go Back

Full Screen

Close

Quit

## Representación de Árbol de Sintaxis Abstracta (cont.)

Los pasos clave para transformar una representación procedimental a una representación de árbol de sintaxis abstracta son:

1. Identificar los constructores de los nuevos valores del tipo y crear un tipo de dato con una variante para cada constructor.
2. Definir los constructores que fabrican las variantes apropiadas de cada tipo de dato.
3. Definir el procedimiento `apply-` para el tipo, usando (`cases type-name ...`) con una cláusula por variante, donde la lista de variables de cada cláusula enumera los parámetros del constructor y la expresión consecuente de cada cláusula es el cuerpo de la expresión lambda.

Home Page

Title Page

Contents



Page 29 of 100

Go Back

Full Screen

Close

Quit

## Representación de Ambientes: Estructura de Datos Alternativa

Gramática:

$$\begin{aligned}\langle \text{env-rep} \rangle &::= () \\ &::= (((\{\langle \text{símbolo} \rangle\}^*) (\{\langle \text{valor} \rangle\}^*)) . \langle \text{env-rep} \rangle)\end{aligned}$$

Una lista vacía representa el ambiente vacío, y una lista no vacía representa un ambiente construido por **extend-env**.

Se puede usar esta definición para implementar la interfaz del ambiente:

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

```
(define extend-env
  (lambda (syms vals env)
    (cons (list syms vals) env)))
```

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 30 of 100

Go Back

Full Screen

Close

Quit

## Representación de Ambientes: Estructura de Datos Alternativa

```
(define apply-env
  (lambda (env sym)
    (if (null? env)
        (eopl:error 'apply-env "No binding for ~s" sym)
        (let ((syms (car (car env)))
              (vals (cadr (car env)))
              (env (cdr env)))
          (let ((pos (rib-find-position sym syms)))
            (if (number? pos)
                (list-ref vals pos)
                (apply-env env sym)))))))))
```

Esta representación es llamada *ribcage*. El ambiente es representado como una lista de listas denominadas *ribs*; el primer elemento de cada *rib* es una lista de símbolos y el segundo elemento del *rib* es la correspondiente lista de valores.

Home Page

Title Page

Contents

◀

▶

◀

▶

Page 31 of 100

Go Back

Full Screen

Close

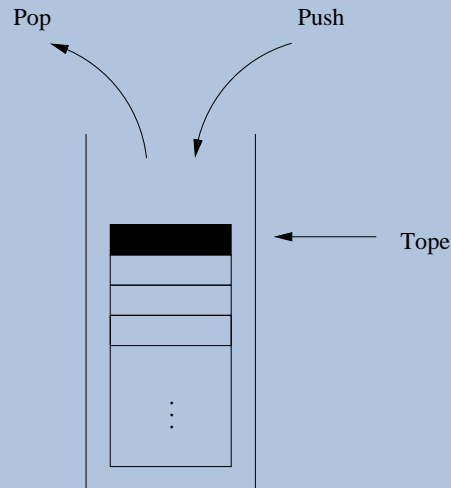
Quit

## Ejemplo 8: Abstracción de Pilas

Una pila es una estructura de datos tipo LIFO (Last-In-First-Out - el último que entra es el primero que sale), la cual es una secuencia de cero o más elementos donde el único elemento “visible” de la estructura es el último que se colocó. Este último elemento es denominado el *tope*.

Dicha estructura es muy utilizada en programación, principalmente para evaluar expresiones, reconocer lenguajes, recorrer árboles y simular procesos recursivos.

La siguiente figura ilustra una pila:



Home Page

Title Page

Contents

◀

▶

◀

▶

Page 32 of 100

Go Back

Full Screen

Close

Quit

## Abstracción de Pilas (cont.)

La interfaz para las pilas consiste de los siguientes procedimientos:



- `(empty-stack)` produce una representación de la pila vacía.
- `(push s v)` produce una nueva pila con los mismos elementos de `s`, en el mismo orden y `v` en el tope de la pila.
- `(pop s)` produce una nueva pila con los mismos elementos de `s`, en el mismo orden, exceptuando el tope, que es eliminado.
- `(top s)` retorna el valor del tope de `s`.
- `(empty-stack? s)` retorna *true* o *false* dependiendo si `s` es vacía o no.





## Abstracción de Pilas (cont.)

La especificación para las operaciones anteriores es entonces:

$$(\text{empty-stack}) = [\emptyset]$$

(push  v) = 

(pop  ) = 

Home Page

Title Page

Contents



Page 34 of 100

Go Back

Full Screen

Close

Quit

## Abstracción de Pilas (cont.)

$$(\text{top } \left[ \begin{array}{c} \boxed{v} \\ \hline \hline \hline \hline \vdots \end{array} \right] ) = v$$

$$(\text{empty-stack? } \left[ \begin{array}{c} \hline \hline \hline \hline \vdots \end{array} \right] ) = \left\{ \begin{array}{ll} \#t & \left[ \begin{array}{c} \hline \hline \hline \hline \vdots \end{array} \right] = [\emptyset] \\ \#f & \left[ \begin{array}{c} \hline \hline \hline \hline \vdots \end{array} \right] \neq [\emptyset] \end{array} \right.$$

Home Page

Title Page

Contents



Page 35 of 100

Go Back

Full Screen

Close

Quit

## Abstracción de Pilas (cont.)

Las operaciones de una pila pueden tener el siguiente código:

```
(define empty-stack
  (lambda ()
    (let ((s '()))
      s)))
```

```
(define push
  (lambda (s v)
    (let ((t (cons v s)))
      t)))
```

```
(define pop
  (lambda (s)
    (let ((t (remove-first (top s) s)))
      t)))
```

```
(define top
  (lambda (s)
    (car s)))
```

```
(define empty-stack?
  (lambda (s)
    (null? s)))
```