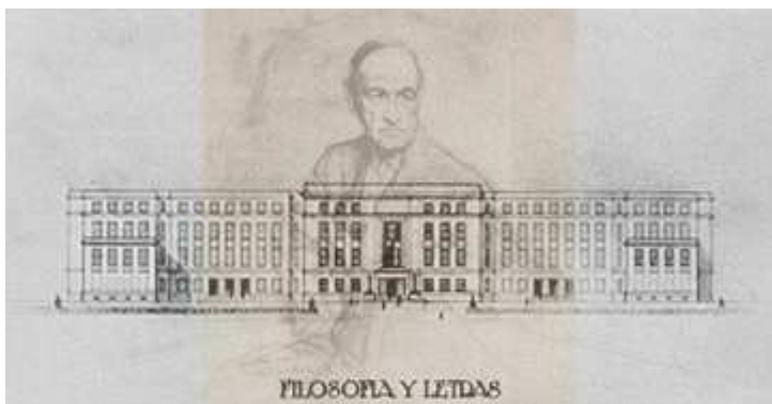


Javier Aroztegui Vélez, Emilio García Buendía y José María Benítez Escario

# Introducción a la programación en Inteligencia Artificial

Antonio Benítez (editor)



Proyecto de Innovación y Mejora de la Calidad Docente n.º 58

Financiado por el Vicerrectorado de Desarrollo y Calidad de la Docencia. Universidad Complutense de Madrid

Madrid, 2009-10



# Índice general

Agradecimientos	IX
<b>I Introducción a la programación</b>	<b>1</b>
<b>1. PLT Scheme</b>	<b>3</b>
1.1. Instalación de PLT Scheme . . . . .	3
1.2. La ventana de DrScheme . . . . .	7
1.2.1. Archivo . . . . .	7
1.2.2. Edición . . . . .	8
1.2.3. Muestra . . . . .	8
1.2.4. Lenguaje . . . . .	8
1.2.5. Scheme . . . . .	9
1.2.6. Insert . . . . .	9
1.2.7. Ventana . . . . .	10
1.2.8. Ayuda . . . . .	11
1.3. Comprobación de la instalación . . . . .	12
<b>2. Introducción a Scheme</b>	<b>13</b>
2.1. Interactuando con Scheme . . . . .	13
2.2. El intérprete sigue reglas . . . . .	15
2.2.1. Forzando la interpretación . . . . .	16
2.2.2. Expresiones compuestas: procedimientos como argumentos .	17
2.3. Las reglas básicas del intérprete . . . . .	19
<b>3. Datos en Scheme</b>	<b>21</b>
3.1. Clases de expresiones de Scheme . . . . .	21
3.1.1. Datos . . . . .	22
3.2. Tipos de datos básicos de Scheme . . . . .	23
3.2.1. Números . . . . .	23
3.2.2. Símbolos . . . . .	24
3.2.3. Valores booleanos . . . . .	25
3.2.4. <i>Strings</i> y caracteres . . . . .	26
3.2.5. Pares y listas . . . . .	27

3.2.6. Vectores . . . . .	30
<b>4. La forma especial <i>Define</i></b>	<b>33</b>
4.1. Variables . . . . .	33
4.2. Procedimientos . . . . .	35
4.2.1. Expresiones <i>lambda</i> . . . . .	35
<b>5. Procedimientos: ejercicios</b>	<b>41</b>
5.1. Primeros ejercicios: procedimientos . . . . .	42
5.1.1. Ejercicios . . . . .	42
5.1.2. Definiciones . . . . .	43
<b>6. Condicionales</b>	<b>47</b>
6.1. La forma especial <i>if</i> . . . . .	47
6.1.1. <i>if</i> anidados . . . . .	48
6.1.2. Acciones múltiples con <i>begin</i> . . . . .	49
6.2. Las formas <i>when</i> y <i>unless</i> . . . . .	50
6.3. La forma especial <i>cond</i> . . . . .	50
<b>7. Procedimientos con condicionales: ejercicios</b>	<b>53</b>
7.1. Un uso de <i>if</i> . . . . .	53
7.2. Diagramas de flujo . . . . .	56
7.3. Uso de <i>ifs</i> anidados . . . . .	56
7.4. Un uso de <i>cond</i> . . . . .	57
7.5. Uso de <i>when</i> y <i>unless</i> . . . . .	60
<b>8. Variables locales</b>	<b>61</b>
8.1. Formas de las variables locales . . . . .	62
8.1.1. Computación en el <i>let-body</i> . . . . .	63
8.1.2. Variantes de <i>let</i> : las variables locales . . . . .	64
8.2. Accesibilidad de variables . . . . .	64
8.2.1. Entornos locales como entornos anidados . . . . .	66
<b>9. Recursión</b>	<b>67</b>
9.1. Componentes de los procesos recursivos . . . . .	68
9.1.1. El rastro de hay-un-impar? . . . . .	69
9.1.2. Ejercicios explicados . . . . .	69
9.1.3. Ejercicios resueltos . . . . .	74
<b>10. Un primer programa</b>	<b>81</b>
10.1. Una calculadora . . . . .	81
10.1.1. Sumadora . . . . .	82
10.1.2. Una primera versión del programa . . . . .	84
10.1.3. Segunda versión del programa con <i>letrec</i> . . . . .	86
10.1.4. Calculadora, una ampliación . . . . .	88

**II Inteligencia Artificial 91**

**11. Programa conexionista o subsimbólico 93**

11.1. Introducción . . . . .	93
11.2. Antecedentes históricos . . . . .	94
11.3. Descripción de un Sistema Neuronal Artificial . . . . .	96
11.3.1. Neuronas artificiales . . . . .	96
11.3.2. Capas de neuronas . . . . .	103
11.4. Aprendizaje en el programa conexionista . . . . .	105
11.4.1. Concepto de aprendizaje . . . . .	105
11.4.2. Aprendizaje supervisado . . . . .	106
11.4.3. Aprendizaje no supervisado . . . . .	107
11.4.4. Aprendizaje por refuerzo . . . . .	108
11.5. Construyendo una red . . . . .	108



# Índice de figuras

1.1. Sitio web de PLT Scheme . . . . .	4
1.2. PLT Scheme según S. O. . . . .	4
1.3. Ventana inicial de DrScheme . . . . .	5
1.4. Límite de memoria . . . . .	5
1.5. Tipo de lenguaje de DrScheme . . . . .	6
1.6. DrScheme configurado . . . . .	6
1.7. Menú básico de DrScheme . . . . .	7
1.8. Ítem <i>Archivo</i> del Menú básico . . . . .	7
1.9. Ítem <i>Edición</i> del Menú básico . . . . .	8
1.10. Buscar–Reemplazar . . . . .	8
1.11. Ítem <i>Muestra</i> del Menú básico . . . . .	9
1.12. Ítem <i>Lenguaje</i> del Menú básico . . . . .	9
1.13. Ítem <i>Scheme</i> del Menú básico . . . . .	10
1.14. Ítem <i>Insert</i> del Menú básico . . . . .	10
1.15. Ítem <i>Ventana</i> del Menú básico . . . . .	11
1.16. Ítem <i>Ayuda</i> del Menú básico . . . . .	11
1.17. Comprobación de la instalación . . . . .	12
3.1. Secuencias de pares . . . . .	29
4.1. Izquierda: texto del programa. Derecha: Intérprete . . . . .	34
4.2. Texto del programa con set! . . . . .	35
5.1. Suma3 . . . . .	41
5.2. Suma3 con + . . . . .	42
5.3. Rectángulo concreto . . . . .	43
5.4. Área de cualquier rectángulo . . . . .	44
5.5. Área de cualquier triángulo . . . . .	44
5.6. Cuadrado de un binomio . . . . .	45
7.1. Programa de saludo . . . . .	55
7.2. Figuras en diagramas de flujo . . . . .	56
7.3. dato-del-tipo . . . . .	59

10.1. Calculadora . . . . .	81
11.1. Ejemplo de una neurona artificial . . . . .	97
11.2. Ejemplo neurona artificial con sus funciones . . . . .	102
11.3. Perceptrón multicapa . . . . .	104
11.4. Construyendo una red . . . . .	108

# Índice de cuadros

11.1. Funciones de propagación . . . . .	98
11.2. Funciones de activación . . . . .	99
11.3. Funciones de salida . . . . .	100



# Agradecimientos

Como responsable del Proyecto de Innovación y Mejora de la Calidad Docente, número 58, quiero expresar mi agradecimiento, en primer lugar, al Vicerrectorado de Desarrollo y Calidad de la Docencia de la Universidad Complutense su apoyo y financiación para el desarrollo de este trabajo.

En segundo lugar, agradezco el magnífico trabajo de los miembros del equipo de investigación: Javier Aroztegui Vélez (profesor Asociado de la UCM), Luis Fernández Moreno (profesor Titular de la UCM), Andrés Rivadulla Rodríguez (profesor Titular de la UCM), José María Benítez Escario (becario predoctoral UCM) y Emilio García Buendía (estudiante de doctorado de la UCM).

Y en tercer lugar, agradezco también el apoyo de la Facultad de Filosofía de la Universidad Complutense.

Madrid, 18 de noviembre de 2010  
Antonio Benítez



Parte I

# Introducción a la programación



# Unidad 1

## PLT Scheme

Scheme<sup>1</sup> es un dialecto de Lisp, de una familia de lenguajes de programación inventados para tratar con palabras, frases e ideas en vez de tener que ver sólo con números. A veces se les llama lenguajes simbólicos.

### Objetivos de esta unidad:

1. Instalar PLT Scheme, después de descargarlo.
2. Familiarizarse con la ventana de DrScheme.

### 1.1. Instalación de PLT Scheme

PLT Scheme es una implementación del lenguaje de programación Scheme<sup>2</sup>. Es un programa que funciona bajo Windows, Mac OS X y Linux. En Google o

---

<sup>1</sup>Fue introducido por Sussman G.J. y Steele G.L. Jr. en 1975 en el MIT, AI Memo n° 349 de título “Scheme: an Interpreter for Extended Lambda Calculus”. El 26 Septiembre del 2007 apareció la revisión 6 de las especificaciones de Scheme. Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten (Editores) han publicado un documento de unas 90 páginas en que se establece el standard: *Revised 6 Report on the Algorithmic Language Scheme*.

<sup>2</sup>En el documento *Guide: PLT Scheme. Version 4.2.3* se dice: «Depending on how you look at it, PLT Scheme is

- a programming language—a descendant of Scheme, which is a dialect of Lisp;
- a family of programming languages—variants of Scheme, and more; or
- a set of tools—for using a family of programming languages.

Where there is no room for confusion, we use simply Scheme to refer to any of these facets of PLT Scheme.

PLT Scheme’s two main tools are

- MzScheme, the core compiler, interpreter, and run-time system; and
- DrScheme, the programming environment (which runs on top of MzScheme)».

Nosotros usaremos DrScheme, configurado de la siguiente manera:

- Scheme/Limit Memory: Unlimited

en cualquier otro buscador puede introducirse «drscheme», lo que llevará a una página de *links*; y eligiendo el *link* PLT Scheme, se llega a la página mostrada en la figura 1.1.

Para descargarlo basta con pulsar en el botón **download**. En la página de descarga basta con desplegar el menú del ítem **Platform:**, seleccionar la versión correspondiente a nuestro sistema operativo y pulsar **download** (figura 1.2).



Figura 1.1: Sitio web de PLT Scheme

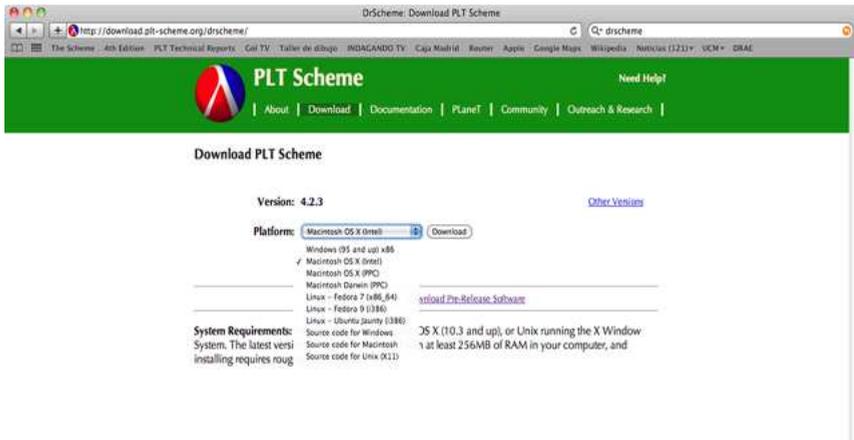


Figura 1.2: PLT Scheme según S. O.

Una vez instalado, hay que lanzar el programa de nombre «DrScheme». Una vez cargado este programa, tendremos una ventana como la de la figura 1.3.

- Lenguaje/Seleccionar lenguaje: Module (la implementación más cercana a  $R^6RS$  según PLT).

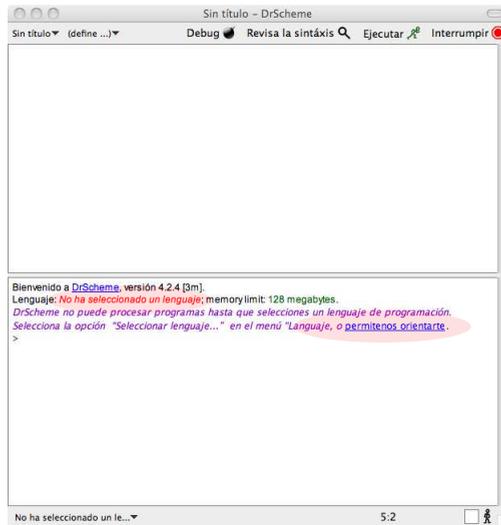


Figura 1.3: Ventana inicial de DrScheme

Antes de usarlo, hay que hacer dos cosas:

- Establecer el límite de memoria en la opción «sin límite» (figura 1.4): Scheme/Limit Memory.



Figura 1.4: Límite de memoria

- Y elegir uno de los tipos de lenguajes que provee DrScheme. Elegiremos «module» (tipo de lenguaje similar al standard R6RS) (figura 1.5): pulsar en permítenos orientarte.

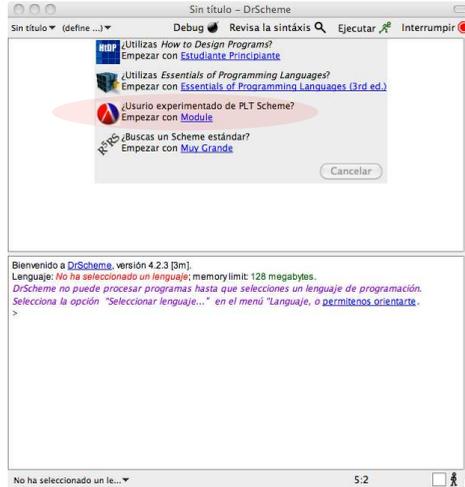


Figura 1.5: Tipo de lenguaje de DrScheme

Después de realizadas estas dos operaciones, la ventana de DrScheme se mostrará como en la figura 1.6.

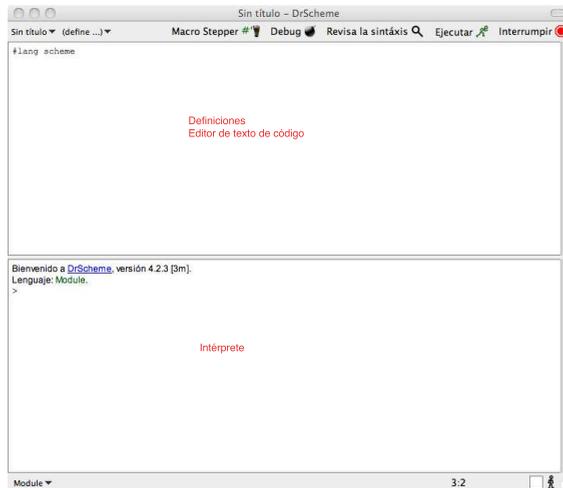


Figura 1.6: DrScheme configurado

## 1.2. La ventana de DrScheme

La ventana de DrScheme tiene cuatro componentes, numerados de arriba hacia abajo los siguientes:

1. una fila de botones;
2. la ventana con la expresión `#lang scheme` (un editor);
3. la ventana del intérprete: Bienvenido a DrScheme, versión 4.2.4 [3m].
4. y, finalmente, una línea de estado donde se muestran diversos mensajes.

Por encima de estos componentes está la línea de ítems del menú básico de DrScheme (figura 1.7);



Figura 1.7: Menú básico de DrScheme

### 1.2.1. Archivo

El menú de nombre «Archivo» ofrece tres grupos de operaciones habituales: abrir ficheros, salvar ficheros e imprimir. Además ofrece tres opciones peculiares de PLT Scheme: abrir una réplica de la ventana de DrScheme (nuevo) o abrir en otra pestaña una nueva ventana del editor, instalar una librería nueva (instalar archivo .plt) y mantener un cuaderno de bitácora con el contenido del editor y de las interacciones con el intérprete. Ver figura 1.8.

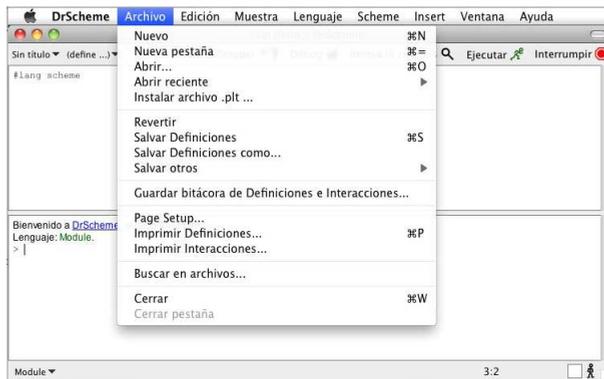


Figura 1.8: Ítem *Archivo* del Menú básico

### 1.2.2. Edición

También el menú de nombre «Edición» ofrece operaciones habituales en otros programas. Quizá lo más peculiar de DrScheme sean las opciones: buscar y modos. Al pulsar buscar se abre una ventana encima de la línea de estado. Dicha ventana puede incorporar otra ventana con el reemplazo deseado (hay que pulsar «show replace»): ver figura 1.10. DrScheme ofrece dos modos del editor: texto y Scheme, que incorpora indentación automática y ajustada a expresiones de Scheme: ver figura 1.9.

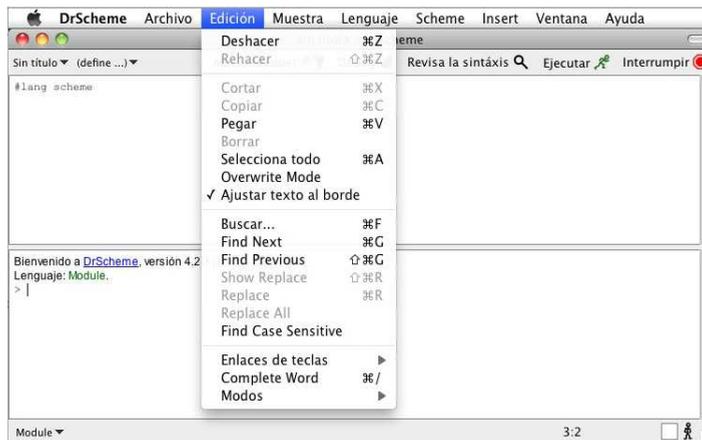


Figura 1.9: Ítem *Edición* del Menú básico

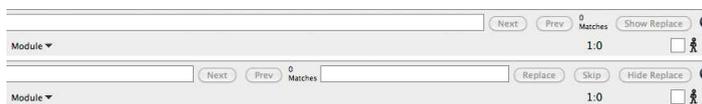


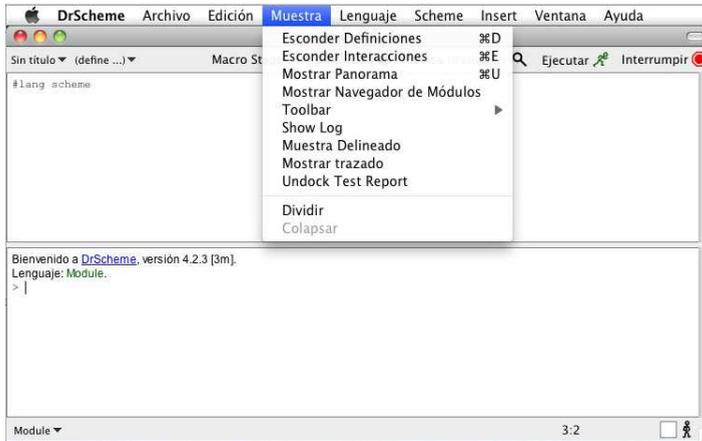
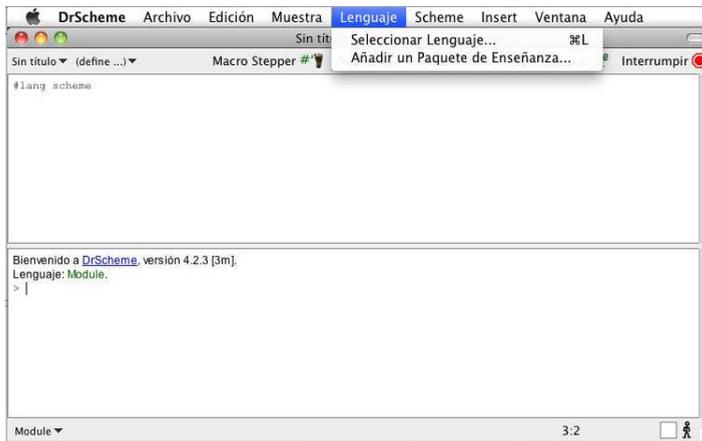
Figura 1.10: Buscar–Reemplazar

### 1.2.3. Muestra

Las dos primeras opciones de este menú sirven para dejar visible o bien la ventana del editor (esconder interacciones) o bien la ventana del intérprete (esconder definiciones). Ver figura 1.11. El resto de opciones pueden no ser usadas.

### 1.2.4. Lenguaje

Las dos únicas opciones son elegir un lenguaje y añadir un paquete de enseñanza (esto sólo es posible con ciertos lenguajes). Ver figura 1.12.

Figura 1.11: Ítem *Muestra* del Menú básicoFigura 1.12: Ítem *Lenguaje* del Menú básico

### 1.2.5. Scheme

Tres opciones son fundamentales: interrumpir la ejecución (ask the program to quit), forzar la terminación de la ejecución y crear un ejecutable. Crear un ejecutable es tanto como crear un *script* ejecutable. Ver figura 1.13.

### 1.2.6. Insert

En este menú encontramos opciones que nos permiten insertar en el texto que estemos escribiendo desde una caja de comentario (un bloque de texto que no será

evaluado por el intérprete) hasta una abreviatura de la palabra *lambda*. Ver figura 1.14.

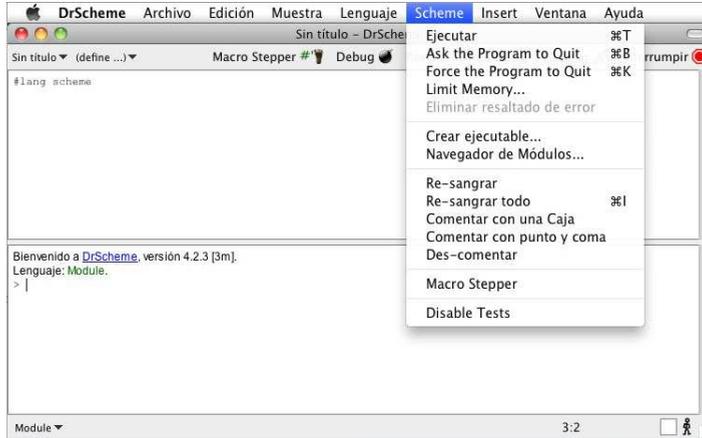


Figura 1.13: Ítem *Scheme* del Menú básico

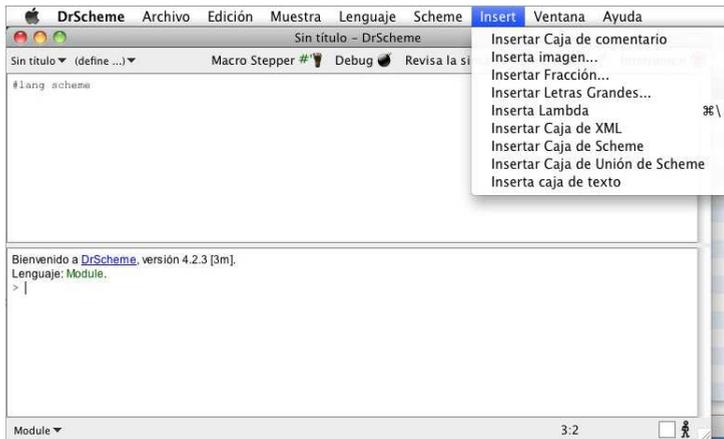


Figura 1.14: Ítem *Insert* del Menú básico

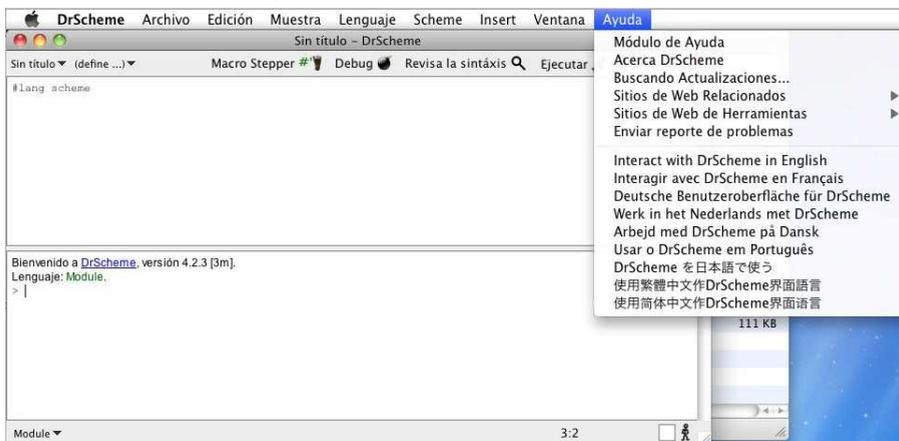
### 1.2.7. Ventana

Además de la posibilidad de hacer zoom, contar con la lista de ventanas abiertas, es posible minimizar y traer al frente una ventana. Ver figura 1.15.

Figura 1.15: Ítem *Ventana* del Menú básico

### 1.2.8. Ayuda

Al seleccionar «Módulo de Ayuda» (figura 1.16), se lanzará el navegador web y conectará con la página inicial de la ayuda de PLT Scheme.

Figura 1.16: Ítem *Ayuda* del Menú básico

### 1.3. Comprobación de la instalación

Para comprobar que la instalación y la configuración de PLT Scheme funciona correctamente, hágase lo siguiente:

1. copiar el siguiente texto en la ventana del editor de texto o definiciones de Scheme:

```
(define intervalo
  (lambda (a b)
    (if (> a b)
        '()
        (cons a
              (intervalo (add1 a) b)))))
```

2. una vez copiado, pulsar el botón Ejecutar
3. escribir en la ventana del intérprete: `(intervalo 1 15)`.

La figura 1.17 muestra todo lo recién enumerado.

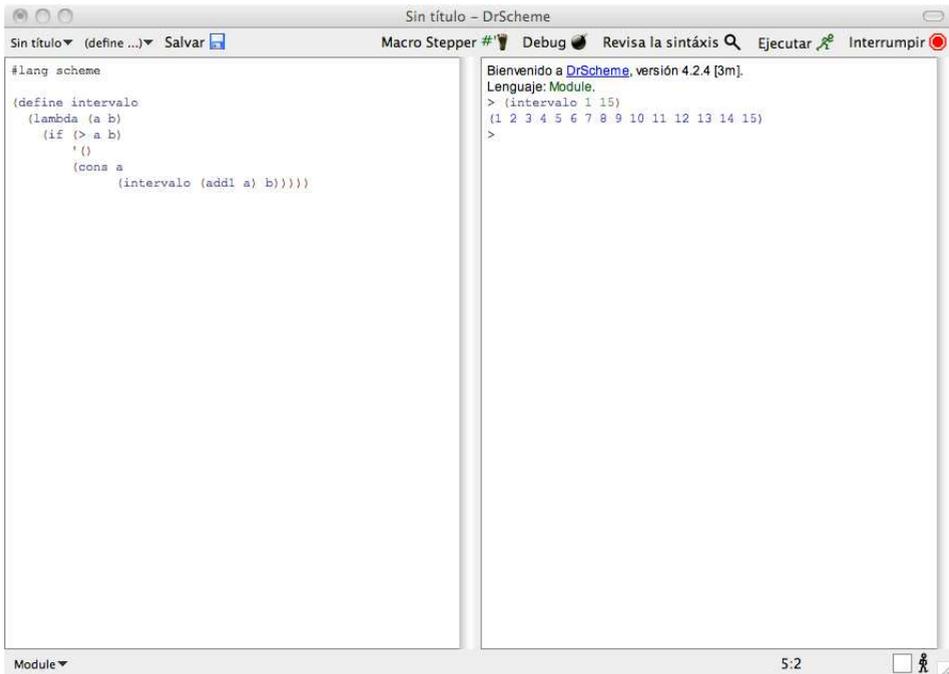


Figura 1.17: Comprobación de la instalación

## Unidad 2

# Introducción a Scheme

En esta unidad nos familiarizaremos con el comportamiento del intérprete, la ventana que contiene:

```
Bienvenido a DrScheme, versión 4.2.4 [3m].  
Lenguaje: Module.  
>
```

El objetivo es comprender que Scheme interpreta las ristas de signos que escribimos según ciertas reglas, reglas que dictan que lo leído sea más que meras ristas de signos.

### 2.1. Interactuando con Scheme

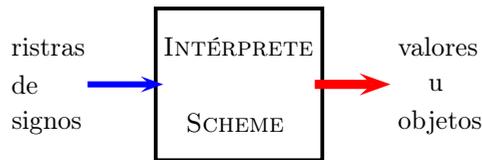
Para escribir un número, hemos de teclear una serie de cifras: a esta ristra de cifras llamamos «número». Scheme lee dicha ristra de cifras y nos devuelve el mismo número que decimos haber escrito. La ristra de cifras siguiente: 123 es el nombre, en notación arábica, del número 123. La ristra de caracteres: CXXIII también designa el número 123, pero en notación romana. Distingamos, pues, entre ristas de signos —lo que tecleamos— y el valor que Scheme nos devuelve una vez que ha interpretado ese *input*.

Si escribimos una ristra de cifras (número) y pulsamos  podemos observar lo que pasa. Igual sucede si escribimos una ristra de letras (palabra) cualquiera.

```
Bienvenido a DrScheme, versión 4.2.4 [3m].  
Lenguaje: Module.  
> 765   
765  
> sevilla   
reference to an identifier before its definition: sevilla
```

- ¿Por qué al escribir un número Scheme nos devuelve el mismo número y, sin embargo, al escribir una palabra nos devuelve un mensaje de error?
- ¿Las palabras y los números son expresiones primitivas para Scheme?
- ¿Qué otras expresiones podemos escribir que Scheme entienda?

Antes de seguir, será bueno fijar un esquema:



- Desde el punto de vista de los *inputs*, de lo que nosotros escribimos (tecleamos) y de lo que Scheme lee, todo *input* es una RISTRA DE SIGNOS de los existentes en el teclado, cada uno perteneciente a una tabla de signos<sup>1</sup>.
- Cada ristra leída por Scheme es, primero, procesada por el intérprete (es lo que representa la caja negra) y, según dicho proceso, Scheme devuelve un valor u objeto<sup>2</sup>.

Podemos introducir más expresiones:

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
...
> 457
457
> 34 + 23
34
#<procedure:+>
23
```

A continuación de 457 hemos escrito la expresión «34 + 23» y la respuesta del intérprete permite observar:

1. que la operación de sumar no se ha producido;
2. que el signo «+» es el nombre de un procedimiento;
3. que los números son entendidos como números.

<sup>1</sup>Una tabla de signos es una lista de pares; cada par incluye el signo y un número o código asociado con él. Se puede manejar el mismo signo con uno u otro de ambos elementos.

<sup>2</sup>Hay distintas clases de valores u objetos. Más abajo las estudiaremos.

Es evidente que algo falta para que se produzca la suma; es decir, la expresión «34 + 23» no induce al intérprete a aplicar el procedimiento de sumar a los números 34 y 23. Por otro lado, la expresión «34 + 23» sigue una *notación infija*: la operación se escribe entre los operandos. Podemos escribir lo mismo en *notación prefija*, así: «+ 34 23».

```
Bienvenido a DrScheme, versión 4.2.4 [3m].
```

```
...
```

```
> + 34 23
```

```
#<procedure:+>
```

```
34
```

```
23
```

De nuevo el intérprete no ejecuta la suma y devuelve el valor de cada sub-expresión. Esto permitirá entender lo siguiente:

- las operaciones se producen en Scheme por aplicación o ejecución de un procedimiento;
- para aplicar un procedimiento hay que escribir la expresión *entre paréntesis*. Así, para que la expresión «+ 34 23» se ejecute es necesario escribirla así: (+ 34 23);
- Scheme sigue una *notación prefija*;
- el esquema general de una expresión que induzca al intérprete a aplicar un procedimiento es el siguiente:

( nombre\_procedimiento argumento<sub>1</sub> argumento<sub>2</sub> ... argumento<sub>n</sub> )

```
Bienvenido a DrScheme, versión 4.2.4 [3m].
```

```
...
```

```
> (+ 34 23)
```

```
57
```

## 2.2. El intérprete sigue reglas

Entendiendo por *expresión una ristra de signos* escritos mediante pulsaciones de teclas, es verdad que Scheme sigue ciertas reglas semánticas o de interpretación de expresiones:

1. Si la expresión leída es un número (una ristra de cifras), devuelve el mismo número. Como en el ejemplo:

```
> 457
```

```
457
```

2. Si la expresión es un símbolo (una ristra de signos en la que al menos un signo no es una cifra), lo interpreta como *un nombre o identificador*. Un nombre o identificador ha de ser el *primer elemento de un par*, cuyo segundo elemento contenga un cierto valor u objeto. Si no existe un par cuyo primer elemento sea el símbolo introducido, devuelve un error. Como en el ejemplo:

```
> sevilla
```

*reference to an identifier before its definition: sevilla.*

En caso contrario, devuelve el valor contenido en el segundo elemento del par.

3. Si la expresión es una lista (una serie de ítems entre paréntesis, separados por un espacio entre sí), la interpreta como la aplicación de un procedimiento u operación. Justamente el procedimiento u operación cuyo nombre es el primer elemento de la secuencia de expresiones. Como en el ejemplo:

```
> (+ 34 23)
```

```
57
```

4. Por último, hay que tener en cuenta que los siguientes signos:

( ) [ ] { } " , ' ' ; # | \

son caracteres especiales para Scheme.

### 2.2.1. Forzando la interpretación

Si queremos escribir un símbolo y no deseamos que Scheme lo interprete como el nombre de una variable, ¿podemos hacer algo? Si queremos escribir una lista y no queremos que Scheme la interprete como la aplicación de un procedimiento, ¿podemos hacer algo? La idea, en ambos casos, es *obligar* a Scheme a interpretar *literalmente* la expresión.

Existe un procedimiento primitivo *quote* que devuelve su argumento evitando las reglas de interpretación. Repárese en que el argumento de *quote* no es interpretado, a diferencia de lo que es normal en otros casos. Por ejemplo:

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: Module.
> (quote sevilla)
sevilla
> (printf "~a ~n" (quote sevilla))
sevilla
> (printf "~a ~n" sevilla)
. . reference to an identifier before its definition: sevilla

```

El *apóstrofo* puede entenderse como una abreviatura de *quote*: evita la interpretación de la expresión que le sigue y la devuelve literalmente. Por ejemplo:

```
> (quote sevilla)
sevilla
> 'sevilla
sevilla
> '(34 56 78)
(34 56 78)
```

En estos últimos ejemplos, `sevilla` es un *literal* y `'(34 56 78)` es una *lista* de datos y *no* la aplicación de un procedimiento.

### 2.2.2. Expresiones compuestas: procedimientos como argumentos

Para sumar `(+ 3 4 7)` lo que Scheme hace es aplicar el procedimiento `«+»` a los argumentos `«3»`, `«4»` y `«7»`. Y si queremos multiplicar el resultado por 5 ¿hemos de interactuar con Scheme como sigue

```
> (+ 3 4 7)
14
> (* 14 5)
70
```

o podemos escribir una única expresión? Y si lo podemos hacer, ¿cómo?

Veamos de más cerca cómo interpreta Scheme la aplicación de un procedimiento: Scheme interpreta la expresión que escribimos de izquierda a derecha de la siguiente manera: lee un `«(»`; comprueba si el último elemento de la expresión es un `«)»`<sup>3</sup>; si lo es, interpreta toda la expresión como la aplicación de un procedimiento.

A continuación lee la siguiente expresión, que ha de ser el nombre de un procedimiento. Sustituye dicho nombre por su valor, esto es la operación en que consiste (por ejemplo, el nombre `«+»` lo sustituye por la operación de sumar).

Pero Scheme no puede aplicar la operación de sumar en el vacío: necesita unos argumentos, unos números que sumar. Una vez leído el nombre del procedimiento y hecha la sustitución, Scheme lee la siguiente expresión en la secuencia y la interpreta según las reglas que hemos visto más arriba, sustituyendo la expresión por el valor resultante de esta interpretación. ¿Qué significa esto? Supongamos que existe una variable global de nombre o identificador `«leo»`, cuyo valor sea `56`<sup>4</sup>. La expresión siguiente es correcta:

```
> (+ 100 leo)
156
```

<sup>3</sup>Los paréntesis han de estar balanceados, es decir, ha de haber tantos `«abre-paréntesis»` como `«cierra-paréntesis»`.

<sup>4</sup>Con la expresión `«(define leo 56)»` se crea una variable global en el intérprete. Más adelante veremos esta idea con más detalle.

¿Por qué? Lo que Scheme hace al interpretar esa expresión puede esquematizarse como sigue:

1. ( ... ) ¿es una lista?
2. Si es una lista aplica el *procedimiento* cuyo identificador es «+».
3. Interpreta cada argumento:
  - a) 100  $\Rightarrow$  100
  - b) leo —un identificador, luego lo sustituye por su valor—  $\Rightarrow$  56.

Si aplicamos esta misma idea, podemos contestar a la pregunta planteada más arriba afirmativamente. En efecto, podemos escribir:

> (\* (+ 3 4 7) 5)  
70

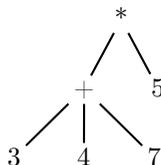
Abstractamente podemos representarnos cómo interpreta Scheme esta expresión:

Aplica procedimiento multiplicar a  $\Leftarrow$  *aplazado hasta que*

Aplica procedimiento sumar a 3 y 4 y 7  $\Leftarrow$  *se evalúa primero*  
y 5.

Es decir, Scheme lleva a cabo dos aplicaciones (o dos operaciones), pero suspende la primera, realiza la segunda y sólo cuando ha acabado ésta, continúa realizando la primera.

Podemos utilizar una representación arbórea para ilustrar esta idea. La expresión «(\*(+ 3 4 7) 5)» puede representarse en un árbol como sigue:



### 2.3. Las reglas básicas del intérprete

1. Una ristra de cifras es evaluada como un número. Devuelve el número.
2. Una ristra de signos, en la que al menos un signo no sea una cifra, es evaluada como un nombre o identificador. Devuelve el valor asociado con el identificador.
3. Una lista (una o varias ristas de signos entre paréntesis) es evaluada como la aplicación de un procedimiento, según las sub-reglas siguientes:
  - a) el primer ítem ha de ser el nombre o identificador de un procedimiento (= operación de cómputo).
  - b) el resto de ítems será considerado los argumentos u operandos de dicho procedimiento.
4. *quote* es una forma especial. Se puede abreviar con el *apóstrofo*.
5. *quote* se aplica así: (*quote* expresión).
6. (*quote* expresión) donde *expresión* es un identificador, devuelve *expresión como valor* (= un dato del tipo *símbolo*).
7. (*quote* lista) donde *lista* es una lista, devuelve *lista como valor* (= un dato del tipo *lista*).



## Unidad 3

# Datos en Scheme

¿Qué es Scheme? Un lenguaje de programación. Es un lenguaje que, como el resto de lenguajes de programación, nadie habla. En este sentido es un lenguaje artificial. En cuanto lenguaje artificial requiere un conjunto de signos; con dichos signos se pueden formar ristras de signos.

Además de un conjunto de signos y ristras de signos, Scheme cuenta con una gramática que sirve para establecer clases de ristras. Por ejemplo: cada ristra formada exclusivamente por cifras y, eventualmente, el punto decimal pertenece a la clase de los números. Otro ejemplo: una ristra de signos cuyo primer y último signo sean «comillas» pertenece a la clase de los *strings*. Análogamente, las ristras que comienzan con un abre-paréntesis y acaban con un cierra-paréntesis pertenecen a la clase de las listas. Hemos visto que una ristra de signos, en la que al menos un signo no es una cifra, pertenece a la clase de los símbolos.

### 3.1. Clases de expresiones de Scheme

Ristras de cifras, con o sin punto decimal, son expresiones bien formadas a las que corresponde un número. Los números son *datos* que Scheme maneja, con los que opera y produce *nuevos datos*. La clase de las expresiones que son listas han de ser consideradas de otra manera. En efecto, una lista corresponde a la aplicación de un procedimiento u operación de cálculo, excepto cuando la lista va precedida por *quote* o el *apóstrofo*: en este caso, la lista es un *dato*. Algo análogo pasa con los símbolos: si el símbolo va precedido por *quote* o el *apóstrofo*, es un *literal* y, por ello, un *dato*; si no es así, es un *identificador* o nombre-de otra «cosa».

Estas mínimas consideraciones nos permiten establecer una división básica: Scheme cuenta con *datos*, por un lado, con *procedimientos*, por otro, y, en tercer lugar, con *expresiones reservadas o especiales* o que tienen una función distinta de los datos y los procedimientos.

### 3.1.1. Datos

En Scheme, como en otros lenguajes de programación, hay que distinguir entre datos simples y compuestos —por un lado— y datos primitivos y definidos —por otro—.

#### Primitivos y Definidos:

Números, listas, *strings*, entre otros, son datos que Scheme conoce y forman parte de su gramática. El programador se los encuentra sin tener que hacer nada, excepto seguir la semántica formal de Scheme. Puede usarlos inmediatamente al escribir sus programas. Sin embargo, si necesita un tipo de datos para representar información, por ejemplo, en forma de árbol: una colección de nodos (en cada uno hay información), en que hay ramas, niveles, antecesores y sucesores, etc., tendrá que *definir* el nuevo tipo apoyándose en tipos ya definidos o primitivos.

#### Simple y Compuestos:

Igual que hemos distinguido entre tipos de datos primitivos y definidos, hay que distinguir entre tipos de datos simples y compuestos. Por ejemplo, el *string* “ferrocarril” —signos entre comillas— es un dato compuesto por cada una de las letras de la palabra. Las comillas indican a Scheme que la ristra de signos ha de ser interpretada como una serie de *caracteres* (letras) que conforman un *string*. Otro ejemplo de la distinción entre datos simples y compuestos son las listas. Scheme interpreta una ristra de signos entre paréntesis como una lista. Entre los paréntesis puede haber distintos tipos de datos: números y símbolos, por ejemplo. La lista es un tipo compuesto de datos y sus ítems pueden ser datos simples o compuestos, por ejemplo: (4567 "ferrocarril")

Según lo visto, cada expresión válida para Scheme pertenecerá a un tipo u otro de dato. Por ello, es conveniente familiarizarse, desde el principio, con algunos tipos de datos que se usan frecuentemente para hacer programas. La operación de memorizar tipos de datos es tediosa, pero puede hacerse siguiendo un esquema formal. El recuadro siguiente lo expone:

- **Predicado de tipo.** Su función es servir para determinar si una expresión pertenece al tipo o no.
  - Un predicado de tipo devuelve como valor #t o #f (verdadero o falso). En general llamaremos «predicado» a todo procedimiento que devuelva como valor #t o #f (verdadero o falso).
  - Convencionalmente, los nombres de los predicados acaban con una interrogación.
- **Procedimientos para construir** un dato, en el caso de los tipos compuestos
- **Procedimientos para extraer** elementos, en el caso de los tipos compuestos
- **Procedimientos para operar** con datos del tipo.

## 3.2. Tipos de datos básicos de Scheme

### 3.2.1. Números

Los números son un tipo de dato que existe en Scheme. Por ello, son *datos primitivos*. Y por ser expresiones simples, *datos simples*. Como cualquier otro tipo de dato, existe un procedimiento primitivo que permite determinar si un argumento dado es o no un número: `number?`. Por ejemplo,

```
Bienvenido a DrScheme, versión 4.2.3 [3m].
Lenguaje: Module.
...
> (number? 7658)
#t
```

Además, existen predicados de *sub-tipos*, por ejemplo: `integer?`, `positive?`, etc.

Los números carecen de procedimientos para *construirlos* y también de procedimientos para *extraer* elementos porque son simples. Pero sí existen *procedimientos primitivos*<sup>1</sup> para **operar** con números. A continuación se explican algunos procedimientos primitivos:

---

`(+ 2 4)`  $\Rightarrow$  6.

---

`(+ 2 4 6 9)`  $\Rightarrow$  21.

---

`(+)`  $\Rightarrow$  0.

---

`(- 2 4)`  $\Rightarrow$  -2.

---

`(- 2 4 6 9)`  $\Rightarrow$  -17.

---

`(-)`  $\Rightarrow$  -: expects at least 1 argument, given 0.

---

`(* 2 4)`  $\Rightarrow$  8.

---

`(* 2 4 6 9)`  $\Rightarrow$  432.

---

`(*)`  $\Rightarrow$  1.

---

`(/ 2 4)`  $\Rightarrow$   $\frac{1}{2}$ .

---

`(/ 2 4 6 9)`  $\Rightarrow$   $\frac{1}{108}$ .

---

<sup>1</sup>Más abajo hablaremos de procedimientos. Reténgase ahora que también existe la distinción entre primitivo y definido.

---

`(/)`  $\Rightarrow$  `/`: expects at least 1 argument, given 0.

---

`(add1 n)`  $\Rightarrow n + 1$ .

---

`(sub1 n)`  $\Rightarrow n - 1$ .

---

`(expt b n)`  $\Rightarrow b^n$ .

---

`(sqrt n)`  $\Rightarrow \sqrt{n}$ .

---

`(quotient 5 2)`  $\Rightarrow 2$ .

---

`(remainder 5 2)`  $\Rightarrow 1$ .

---

`(random 8)`  $\Rightarrow$  al azar un número entre 0 y 7.

---

`(> 7 3)`  $\Rightarrow$  `#t`.

---

`(< 7 3)`  $\Rightarrow$  `#f`.

---

`(= 7 3)`  $\Rightarrow$  `#f`.

---

`(number? 43)`  $\Rightarrow$  `#t`.

---

`(integer? 43)`  $\Rightarrow$  `#t`.

---

`(even? 43)`  $\Rightarrow$  `#f`.

---

`(odd? 43)`  $\Rightarrow$  `#t`.

---

`(number->string 45)`  $\Rightarrow$  `"45"`.

---

`(integer->char 45)`  $\Rightarrow$  `#\-`.

---

`(char->integer #\A)`  $\Rightarrow$  65.

### 3.2.2. Símbolos

Una ristra de signos en la que al menos un signo *no es* una cifra, es un símbolo. Hay que recordar la regla semántica básica de Scheme según la cual un símbolo es interpretado como un identificador o nombre. Y que la expresión `(quote ristra)` convierte a *ristra* en un símbolo o literal.

Existe un predicado de tipo para símbolos: `symbol?`, que devuelve `#t` o `#f` según que su argumento sea o no un símbolo o literal. El predicado `eq?` determina

si dos símbolos son o no el mismo. Además de estos predicados, se cuenta con dos procedimientos para convertir entre tipos de datos:

---

`(symbol? expr) ⇒ #t`, si `expr` es un literal o un identificador cuyo valor es un literal.

---

`(eq? expr1 expr2) ⇒ #t`, si `expr1` y `expr2` son o se refieren al mismo literal.

---

`(symbol->string expr) ⇒ "expr"`.

---

`(string->symbol string) ⇒` el literal formado por los signos de `string`.

### 3.2.3. Valores booleanos

El conjunto de valores veritativos o booleanos está formado por dos elementos  $\{\#t, \#f\}$ . Las operaciones, que dependen de un valor veritativo, consideran cualquier objeto distinto de `#f` como `#t`. Por ejemplo, `(= 34 75)` es una expresión correcta que establece si dos números dados son o no iguales. Por tanto, el valor que dicha expresión devuelve será o `#t` o `#f`. Pero `member` es un procedimiento parecido a un predicado porque devuelve `#f` si su primer argumento no es un elemento del segundo (una lista), por ejemplo: `(member 2 '(a e i o u))` devuelve `#f`. Y, sin embargo, `(member 'i '(a e i o u))` devuelve la sublista `(i o u)`. En tanto que `(i o u)` es distinto de `#f`, tendrá el mismo efecto que el valor veritativo `#t`. El predicado de tipo es `boolean?`.

#### Negación:

---

`(not #f) ⇒ #t`.

---

`(not #t) ⇒ #f`.

---

`(not "Hola") ⇒ #f`.

#### Conjunción:

---

`(and #t #t ... #t) ⇒ #t`.

---

`(and #t #t ... #f) ⇒ #f`.

---

`(and) ⇒ #t`.

---

`(and "Hola" 2) ⇒ 2`.

**Disyunción:**


---

 $(\text{or } \#f \ \#f \ \dots \ \#f) \Rightarrow \#f.$ 


---

 $(\text{or } \#f \ \#t \ \dots \ \#f) \Rightarrow \#t.$ 


---

 $(\text{or}) \Rightarrow \#f.$ 


---

 $(\text{or } \text{"Hola"} \ 2) \Rightarrow \text{"Hola"}.$ 
**Igualdad:**


---

 $(\text{eq? } v1 \ v2) \Rightarrow \#t, \text{ si, y sólo si, } v1 \text{ y } v2 \text{ denotan el mismo objeto.}$ 


---

 $(\text{eqv? } v1 \ v2) \Rightarrow \#t, \text{ si se cumple que } (\text{eq? } v1 \ v2) \Rightarrow \#t.$ 


---

 $(\text{equal? } v1 \ v2) \Rightarrow \#t, \text{ si se cumple que } (\text{eqv? } v1 \ v2) \Rightarrow \#t.$ 
**3.2.4. Strings y caracteres**

Una ristra de signos entre comillas es un *string*. Por ejemplo: ‘123456’, ‘asdfg’ o ‘zaragoza-123’. Cada uno de los signos que aparece en un *string* es un **carácter**. Por tanto, cada *string* está compuesto de caracteres: *string* es un tipo de datos *compuesto*, mientras que los caracteres son un tipo de datos *simple*. Pero caracteres y *strings* son tipos de datos primitivos. Lo opuesto a un tipo de datos primitivo es un tipo de datos *definido*.

El predicado de tipo correspondiente a *strings* es: **string?**. Y el correspondiente a caracteres es: **char?**.

Para introducir el carácter «a» no basta con teclear el signo «a». Al teclear el signo «a» (o más formalmente: al introducir la ristra cuyo único signo es «a»), Scheme lo interpreta como un símbolo. Para que Scheme lo interprete como el carácter «a» hay que escribir la ristra `#\a`, lo que se conoce como la representación externa del carácter «a».

Hay otros predicados que permiten determinar si dos caracteres son iguales, mayor o menor uno que otro, según su ordenación en una tabla de signos: **char=?**, **char>?**, **char<?**. Además, puede convertirse un carácter dado en el entero correspondiente a su código y al contrario, por ejemplo: **(char->integer #\a)**, **(integer->char 67)**.

Más interesante que los caracteres son los *strings*. En efecto, las operaciones posibles como unir o concatenar uno o varios *strings*, extraer una porción de un *string* dado, comparar dos *strings*, etc., tienen su correspondiente procedimiento. A continuación se explican algunos procedimientos primitivos:

---

constructor:

(`make-string` *n* <*char*>) ⇒ un *string* de *n* lugares. Si se proporciona un carácter, cada lugar del *string* es rellenado con *char*.

---

constructor:

(`string` #\a #\b #\w) ⇒ el *string* “abw”.

---

extractor:

(`string-ref` *string* *índice*) ⇒ el carácter cuyo lugar corresponde al número de orden *índice* (el índice del primer carácter es 0).

---

extractor:

(`substring` *string* *inicio* <*final*>) ⇒ el trozo de *string* que comienza en el lugar *inicio* y que llega hasta el lugar *final*. Si *final* no es provisto, se toma el último lugar del *string*.

---

operaciones:

(`string-append` “*string-1*” “*string-2*” ... “*string-n*”) ⇒ “*string-1string-2...string-n*”.

---

operaciones:

(`string-length` “*string*”) ⇒ número de lugares con que cuenta *string*.

---

operaciones:

(`string-set!` *string* *índice* *new-char*) ⇒ establece *new-char* como el valor del lugar correspondiente a *índice*.

---

comparaciones:

(`string=?` *string1* *string2*) ⇒ #t, si ambos *strings* son idénticos.

---

comparaciones:

(`string<?` *string1* *string2*) ⇒ *string1* es menor que *string2*, si (`char<?` *char*<sub>0-1</sub> *char*<sub>0-2</sub>) = #t.

---

comparaciones:

(`string>?` *string1* *string2*) ⇒ *string1* es mayor que *string2*, si (`char>?` *char*<sub>0-1</sub> *char*<sub>0-2</sub>) = #t.

---

### 3.2.5. Pares y listas

Scheme es un lenguaje que deriva de Lisp (lenguaje para el procesamiento de listas). Por ello, las listas son uno de los tipos de datos que más se usan para escribir programas. En Lisp, las listas estaban construidas como secuencias finitas de pares. Y cada par estaba constituido por dos elementos que recibían los nombres

de `car` (léase: car) y `cdr` (léase: culder). Como las listas, un par se representa como dos ítems, separados por un punto, entre paréntesis, por ejemplo: `(1 . 2)`.

Una característica de Lisp, que Scheme heredó en sus primeras versiones, era el carácter dinámico de los pares: tanto el `car` como el `cdr` de un par podían cambiar su valor, su contenido, cuando un programa se estaba ejecutando. Si el par en cuestión formaba parte de una lista, dicho cambio dinámico en el par se reflejaba en la lista que lo contuviera.

A partir de la revisión 6.<sup>a</sup> del Report, se distinguen dos clases de pares: pares inmutables, o que no pueden cambiar dinámicamente, y pares mutables, que sí pueden cambiar dinámicamente.

### Pares inmutables

La representación externa de un par inmutable es `(car . cdr)`. El predicado de tipo es `pair?`.

---

constructor:

`(cons obj1 obj2) ⇒ (obj1 . obj2)`.

---

extractor:

`(car (cons 'a 1)) ⇒ a`.

---

extractor:

`(cdr (cons 'a 1)) ⇒ 1`.

### Listas formadas por pares inmutables

Una lista es una serie de ítems entre paréntesis. Un caso excepcional es la lista vacía —de nombre `empty`—: `'()`. Las listas son, pues, un tipo de dato *compuesto*. Incluso, una lista puede albergar otras listas entre sus elementos, por ejemplo: `'(1 a (d f g) z x (q (w)))`.

El predicado de tipo es `list?`. El uso combinado de `pair?` y `list?` obliga a entender que no es lo mismo el aspecto externo de una lista y su forma interna.

```

Bienvenido a DrScheme, versión 4.2.3 [3m].
Lenguaje: Module.
> (list? (cons 1 2))
#f
> (pair? (list 1 2 3))
#t

```

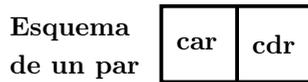
La lista `(1 2 3)` es reconocida como un par, pero el par `(1 . 2)` no es reconocido como una lista. ¿Por qué? Porque *internamente* una lista es una *secuencia finita de pares*, mientras que un par es simplemente un par. Pero ¿qué pasa con los pares compuestos? Por ejemplo:

```

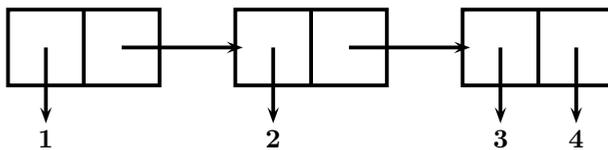
Bienvenido a DrScheme, versión 4.2.3 [3m].
Lenguaje: Module.
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
> (list? (cons 1 (cons 2 (cons 3 4))))
#f
    
```

Las secuencias de pares son **listas** (propias) si, y sólo si, el `cdr` del último par está ocupado por el objeto **empty** (la lista vacía). En otro caso, son simples secuencias de pares (o *listas impropias*). Es lo que quiere ilustrar la figura 3.1. De modo que una secuencia finita de pares puede ser:

- una **lista propia**: si el `cdr` del último par está ocupado por *la lista vacía*.
- una **lista impropia**: si el `cdr` del último par *no* está ocupado por la lista vacía.



$(\text{cons } 1 (\text{cons } 2 (\text{cons } 3 4))) \Rightarrow (1\ 2\ 3\ .\ 4)$



$(\text{list } 1\ 2\ 3) \Rightarrow (1\ 2\ 3)$  o bien:

$(\text{cons } 1 (\text{cons } 2 (\text{cons } 3 \text{ empty}))) \Rightarrow (1\ 2\ 3)$

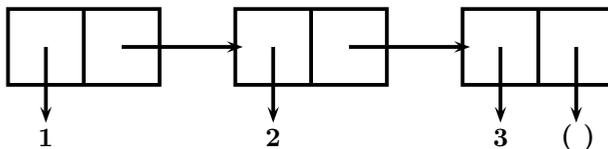


Figura 3.1: Secuencias de pares

---

predicado de tipo:

`(list? obj) ⇒ #t`, si `obj` es una lista.

---

constructor:

`(list obj1 obj2 ... objn) ⇒ (obj1 obj2 ... objn)`.

---

constructor/modificador:

`(cons obj lista) ⇒` otra lista cuyo primer elemento es `obj` y el resto todos los de `lista`.

---

extractor:

`(car lista)` o `(first lista)` ⇒ el primer elemento de `lista`.

---

extractor:

`(cdr lista)` o `(rest lista)` ⇒ la sublista resultante de quitar el primer elemento de `lista`.

---

extractor:

`(list-ref lista índice) ⇒` el elemento que ocupa el lugar correspondiente a `índice` (el índice del primer elemento es 0).

---

operaciones:

`(length lista) ⇒` el número total de ítems que hay en `lista`.

---

operaciones:

`(append lista1 lista2 ... listan) ⇒` una lista conteniendo todos los elementos de `lista1 lista2 ... listan` en este orden.

---

operaciones:

`(list-tail lista k) ⇒` la sublista que resulta de eliminar los  $k$  primeros elementos de `lista`.

---

operaciones:

`(take lista k) ⇒` la sublista con los  $k$  primeros elementos de `lista`.

---

### 3.2.6. Vectores

Un vector es un tipo de dato compuesto cuyos elementos pueden ser heterogéneos. Como en las listas, un vector puede tener vectores como ítems. Como las listas y los *strings*, los elementos de un vector están ordenados. La representación del vector cuyos elementos son: 1 #\a 2 #\b 3 #\c, es

$$\#(1 \#\backslash\text{a} 2 \#\backslash\text{b} 3 \#\backslash\text{c}).$$

La longitud de un vector es fija. El predicado de tipo es `vector?`.

---

predicado de tipo:

(vector? obj)  $\Rightarrow$  #t, si obj es un vector.

---

constructor:

(vector obj1 obj2 ... objn)  $\Rightarrow$  #(obj1 obj2 ... objn).

---

constructor:

(make-vector n <obj>)  $\Rightarrow$  un *vector* de *n* lugares o ítems. Si se proporciona un objeto, cada lugar del *vector* es rellenado con obj.

---

modificador:

(vector-set! vector i obj)  $\Rightarrow$  cambia el *i-ésimo* elemento de vector por obj.

---

extractor:

(vector-ref vector i)  $\Rightarrow$  devuelve el elemento *i-ésimo* de vector. El índice del primer elemento es 0..

---

operaciones:

(vector-length vector)  $\Rightarrow$  devuelve el número de elementos de vector.

---

operaciones:

(equal? vector1 vector2)  $\Rightarrow$  devuelve #t si vector1 y vector2 tienen igual longitud y los mismos elementos en las mismas posiciones.

---

operaciones:

(vector->list vector)  $\Rightarrow$  devuelve una lista con los mismos elementos, y en las mismas posiciones, de vector.

---

operaciones:

(list->vector lista)  $\Rightarrow$  devuelve una vector con los mismos elementos, y en las mismas posiciones, de lista.



## Unidad 4

# La forma especial *Define*

En la sección 3.1 hemos visto que existen tres clases de expresiones: datos, procedimientos y expresiones o formas especiales. **Define** pertenece a estas últimas: es una forma especial, si bien su aplicación parece la de un procedimiento, porque se usa según el esquema siguiente:

```
(define identificador valor)
```

En el lugar ocupado por *valor* puede escribirse o un dato o una *expresión lambda* (procedimiento).

- Si se escribe un dato, **define** crea una variable.
- Si se escribe una *expresión lambda*, **define** crea un nuevo procedimiento.

### 4.1. Variables

Un ejemplo del uso de **define** para crear una variable es el siguiente:

```
Bienvenido a DrScheme, versión 4.2.4 [3m].  
Lenguaje: scheme.  
> (define sevilla 567)  
> sevilla  
567  
> (+ sevilla 1)  
568
```

Una vez que hemos creado una variable, podemos cambiar su valor. O dicho de otra manera: una variable permanece aunque su valor cambie. En el ejemplo anterior, la variable «sevilla» sigue siendo la misma variable aún cuando su valor cambiara a 100, por ejemplo.

¿Cómo cambiamos el valor de una variable? Mediante `set!`, que es otro procedimiento primitivo. Como `define`, `set!` es también una *forma especial* porque no interpreta su primer argumento<sup>1</sup> aunque sí su segundo. Su sintaxis es muy simple:

```
(set! identificador nuevo-valor)
```

Por último, hay que tener en cuenta que una variable definida en un programa será entendida por el intérprete como una *constante inmodificable*, a no ser que dicho programa cambie el valor de dicha variable mediante `set!`.

La figura 4.1 ilustra lo que sucede cuando no existe una expresión con `set!` en el programa:

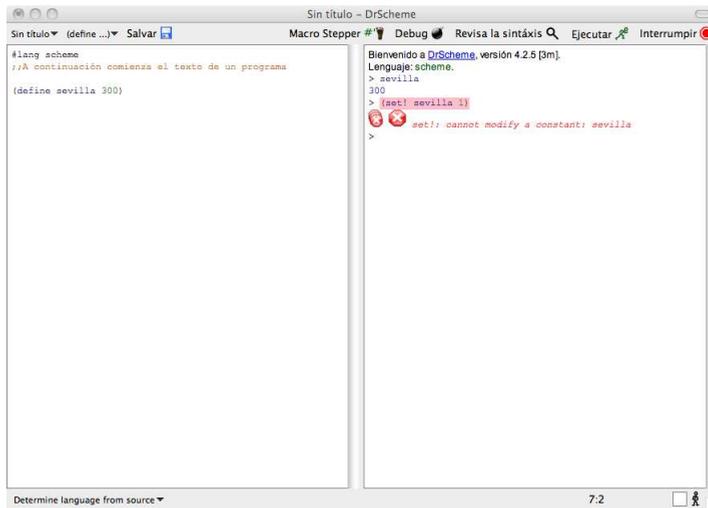


Figura 4.1: Izquierda: texto del programa. Derecha: Intérprete

Y la figura 4.2 ilustra lo que sucede cuando sí existe una expresión con `set!` en el programa:

<sup>1</sup>Aunque si el nombre o identificador dado no lo es de una variable existente, se produce un error. Por ejemplo: `(set! leon 40)` devolverá el mensaje: `set!: cannot set identifier before its definition: leon`, si `leon` no es el nombre de una variable definida ya.

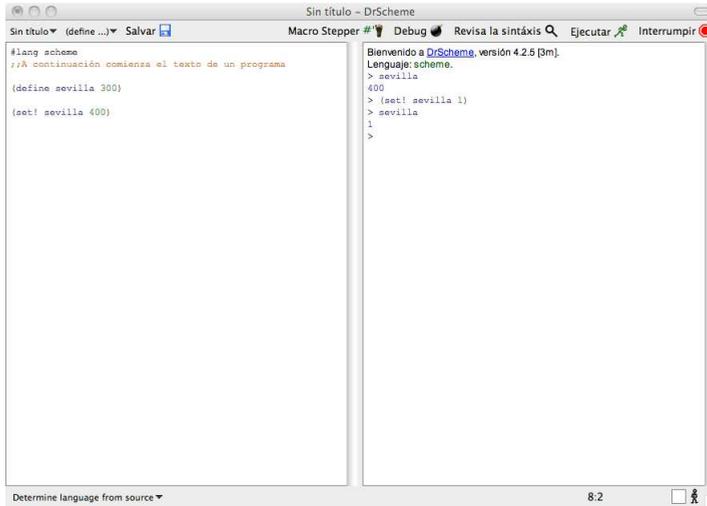


Figura 4.2: Texto del programa con set!

## 4.2. Procedimientos

La forma especial `define` crea un nuevo procedimiento cuando su valor es una *expresión lambda*. ¿Qué es una expresión *lambda*?

### 4.2.1. Expresiones *lambda*

Cada expresión que sigue el siguiente esquema:

```
(lambda
  argumentos
  instrucción-1
  instrucción-2
  ...
  instrucción-n)
```

es la definición de un procedimiento. Veamos más de cerca este esquema.

1. Es una lista cuyo primer elemento es la forma especial «lambda».
2. En dicha lista hay otros dos elementos más:
  - a) El que se llama en el esquema «argumentos». Su posición, segunda, es fija.
  - b) La secuencia de líneas llamadas «instrucción-1», «instrucción-2», ..., «instrucción-n» forman una unidad que recibe el nombre de *lambda-body*. Su posición, también fija, es la tercera.

**Los argumentos de *lambda*:**

Los argumentos de `lambda` admiten cuatro formas de expresión:

- `()`. El único argumento de `lambda` es la lista vacía. Esto significa que el procedimiento creado mediante la expresión `lambda` no admite, al ser aplicado, ningún argumento.
- `(arg-1 arg-2 ... arg-n)`. En este caso, el procedimiento creado *exige*, al ser aplicado, tantos argumentos cuantos nombres se hayan introducido en la lista de los argumentos `lambda`.
- `(arg-1 arg-2 ... arg-n . optional)`. En esta forma, `lambda` crea un procedimiento que, al ser aplicado, exige tantos argumentos cuantos nombres haya antes del punto y, además, acepta un número indeterminado más de argumentos que internamente serán tratados como una lista.
- *optional*. En este caso, la expresión `lambda` no tiene una lista de argumentos sino un nombre que designa un argumento único, si bien el procedimiento, al ser aplicado, puede tener diversos argumentos los cuales serán tratados internamente como una lista.

Para entenderlo mejor y prácticamente vamos a escribir un ejemplo de procedimiento en el que sólo varíen los argumentos. Como única instrucción usaremos la siguiente: `(printf 'mensaje: sin argumentos ~n')`.

`Printf` es un procedimiento primitivo, dado en el intérprete de Scheme. Para aplicarlo se escribe una expresión según el siguiente esquema

```
(printf un-string <argumentos>)
```

donde:

- `un-string`: ha de ser *un string*, algo entre comillas.
- `<argumentos>`: si se usan, exige incorporar en el *string* la expresión «`~a`» tantas veces cuantos sean los argumentos mencionados.
- además de la expresión «`~a`», puede usarse la expresión «`~n`» para añadir un avance de línea.

Ejemplos de definición de un procedimiento:

1. `lambda` con la lista vacía como argumento.

```
(define primer-proc
  (lambda ()
    (printf "mensaje: sin argumentos~n")
  ))
```

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (primer-proc)
mensaje: sin argumentos

```

2. `lambda` con dos argumentos (que podrían ser más).

```

(define primer-proc
  (lambda (a b)
    (printf "mensaje: exige 2 argumentos; son: ~a, ~a~n" a b)
  ))

```

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (primer-proc 1 'dedo)
mensaje: exige 2 argumentos; son: 1, dedo

```

3. `lambda` con dos argumentos y uno más opcional.

```

(define primer-proc
  (lambda (a b . c)
    (printf "mensaje: exige 2 argumentos y puede tener
otro más opcional; son: ~a, ~a, ~a~n" a b c)
  ))

```

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (primer-proc 1 'dedo 9876)
mensaje: exige 2 argumentos y puede tener otro más opcional;
son: 1, dedo, (9876)
> (primer-proc 1 'dedo 9876 'león)
mensaje: exige 2 argumentos y puede tener otro más opcional;
son: 1, dedo, (9876 león)
> (primer-proc 1 'dedo)
mensaje: exige 2 argumentos y puede tener otro más opcional;
son: 1, dedo, ()

```

4. `lambda` con un único argumento opcional.

```

(define primer-proc
  (lambda opcional
    (printf "mensaje: opcional es una lista: ~a~n" opcional)
  ))

```

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (primer-proc 1 'dedo 9876 'león)
mensaje: opcional es una lista: (1 dedo 9876 león)
> (primer-proc)
mensaje: opcional es una lista: ()

```

### El cuerpo de *Lambda* o *lambda-body*

Normalmente, el cuerpo de una *expresión lambda* está formado por otras expresiones que el intérprete ha de reconocer. Si el cuerpo de la expresión lambda no contiene ninguna expresión, se produce un error. Por ejemplo:

```

(define lambda-body
  (lambda (a)
    ))

```

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
lambda: bad syntax in: (lambda (a))

```

Si el cuerpo de la *expresión lambda* está formado por varias expresiones correctas, todas ellas forman una secuencia ordenada de expresiones. Por ello, el proceso computacional, que se produce al aplicar el procedimiento, puede cambiar, si se altera el orden de las expresiones.

Veamos esta idea en un ejemplo. Sea la definición de un procedimiento la siguiente:

```

(define lambda-body
  (lambda (a b)
    (printf "el primer argumento es: ~a~n" a)
    (printf "el segundo argumento es: ~a~n" b)
    (+ a b)
    ))

```

Una vez escrito ese texto en el editor, pulsamos el botón de ejecutar y lo aplicamos en el intérprete:

```

Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (lambda-body 23 43)
el primer argumento es: 23
el segundo argumento es: 43
66

```

Si alteramos el orden de las expresiones y escribimos:

```
(define lambda-body
  (lambda (a b)
    (printf "el segundo argumento es: ~a~n" b)
    (printf "el primer argumento es: ~a~n" a)
    (+ a b)
    ))
```

obtenemos otro resultado:

```
Bienvenido a DrScheme, versión 4.2.4 [3m].
Lenguaje: scheme.
> (lambda-body 23 43)
el segundo argumento es: 43
el primer argumento es: 23
66
```

### El valor devuelto por un procedimiento:

Al aplicar un procedimiento se produce un *proceso computacional* cuyas acciones son determinadas por las expresiones existentes en el *lambda-body*. Las acciones de un proceso computacional pueden ser muy diferentes entre sí, por ejemplo: cambiar el valor de una variable externa o global, aplicar una operación (por ejemplo, sumar dos números), etc. Sin embargo, no en todos los casos, Scheme devuelve un valor. Hemos de distinguir entre dos clases de procedimientos:

- aquellos que devuelven un valor. Es característico que dicho valor pueda ser usado como argumento de otro procedimiento;
- otros que no devuelven un valor. En estos casos, la aplicación del procedimiento no puede figurar como argumento de otro procedimiento.

Por ejemplo:

- `(+ 2 3)` devuelve el valor 5. Y es posible escribir: `(* 3 (+ 2 3))`, procedimiento que multiplica el valor devuelto por `(+ 2 3)` por 3.
- por el contrario `(display (+ 2 3))` no devuelve ningún valor. Por ello, la expresión `(* 3 (display (+ 2 3)))` produce un error.

Bienvenido a DrScheme, versión 4.2.4 [3m].

Lenguaje: scheme.

> (+ 2 3)

5

> (\* 3 (+ 2 3))

15

> (display (+ 2 3))<sup>a</sup>

5

> (\* 3 (display (+ 2 3)))

\*: expects type <number> as 2nd argument, given: #<void>;  
other arguments were: 3

---

<sup>a</sup>Display es una primitiva que escribe su argumento en pantalla y devuelve como valor `void`, que significa «no-valor».

# Unidad 5

## Procedimientos: ejercicios

Tantos los procedimientos como los programas pueden ser entendidos como una tarea que tiene un fin, un objetivo. Por ejemplo, no existe en Scheme un procedimiento primitivo que sume 3 a un número dado. Si quisiéramos contar con un procedimiento que haga la tarea cuyo fin es obtener el valor correspondiente a sumar 3 a cualquier número dado, lo tendríamos que definir. ¿Cómo se puede pensar esa definición? Inicialmente sabemos sólo dos cosas: el nombre que damos a la tarea (supongamos: `suma3`) y que tiene un único argumento (llamémoslo: `num`). Nos lo podemos representar así:

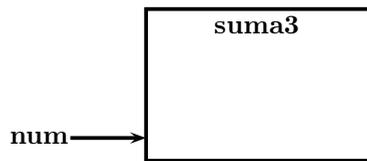


Figura 5.1: Suma3

La tarea de `suma3` consiste en sumar 3 a un número dado (`num`). Por tanto, haremos uso en su definición del procedimiento primitivo `+`. Lo que nos podemos representar así:

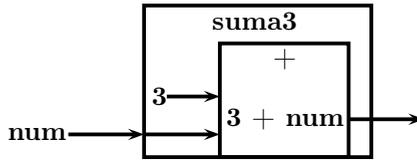


Figura 5.2: Suma3 con +

Y podemos traducir esta representación a la siguiente definición:

```
(define suma3
  (lambda (num)
    (+ 3 num)
  ))
```

### En síntesis

1. Definir un procedimiento, que no es un procedimiento primitivo de Scheme, consiste en pensar una tarea que tiene un objetivo.
2. Dicha tarea ha de contar con la especificación de cuántos argumentos tiene.
3. Y ha de usar procedimientos previamente definidos o existentes en Scheme.

## 5.1. Primeros ejercicios: procedimientos

Vamos a definir cinco procedimientos nuevos. En cada uno seguiremos el siguiente esquema:

1. definir la tarea. Fijando el objetivo y los argumentos;
2. estableceremos los procedimientos existentes con los que hay que contar;<sup>1</sup>
3. escribiremos el código en el editor;
4. y lo probaremos pulsando el botón de ejecutar y aplicando el procedimiento definido.

### 5.1.1. Ejercicios

1. Un procedimiento cuya tarea sea calcular el área de un rectángulo cuya base sea igual a 5 y cuya altura sea 3.<sup>2</sup>

<sup>1</sup>Se aconseja al estudiante que use la técnica de cuadros o bloques de las figuras 5.1 y 5.2

<sup>2</sup>Área de un rectángulo =  $base \times altura$ .

2. Generalizar la anterior tarea tal que sea posible calcular el área de cualquier rectángulo dados su base y su altura.
3. Un procedimiento que permita calcular el área de cualquier triángulo.<sup>3</sup>
4. Un procedimiento que permita calcular el cuadrado de cualquier binomio dado.<sup>4</sup>
5. Este ejercicio consiste no en calcular el cuadrado de un binomio dado, sino en producir la expresión en que se expande  $(a + b)^2$ , dados unos argumentos concretos. Por ejemplo: (cuad-binomio 3 4)  $\Rightarrow$  (+ (expt 3 2) (expt 4 2) (\* 2 3 4))

### 5.1.2. Definiciones

1. Tal como está definida la tarea, su objetivo consiste en calcular el área de un rectángulo concreto cuya base mide 5 y su altura 3. Por tanto, se puede representar así:

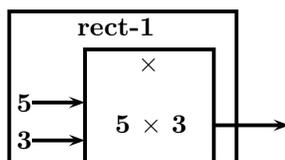


Figura 5.3: Rectángulo concreto

Editor

```
#lang scheme
;; texto en el editor

(define rect-1
  (lambda ()
    (* 5 3)
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (rect-1)
15
```

2. Para generalizar es necesario que el procedimiento tenga dos argumentos (*base* y *altura*). Esta idea se puede representar así:

<sup>3</sup>Área de un triángulo =  $1/2(\text{base} \times \text{altura})$ .

<sup>4</sup>Cuadrado de un binomio:  $(a + b)^2 = a^2 + b^2 + 2ab$ .

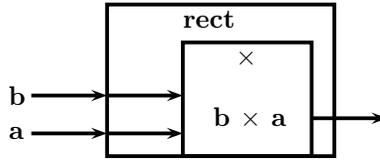


Figura 5.4: Área de cualquier rectángulo

Editor

```
#lang scheme
;; texto en el editor

(define rect
  (lambda (b a)
    (* b a)
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (rect 20 10)
200
```

3. El área de un triángulo puede expresarse así:  $(/ (* \text{base altura}) 2)$ . Se trata de una expresión que compone dos procedimientos, uno como argumento del otro. Esta idea puede representarse así:

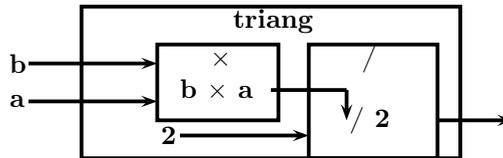


Figura 5.5: Área de cualquier triángulo

Editor

```
#lang scheme
;; texto en el editor

(define triang
  (lambda (b a)
    (/ (* b a) 2)
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (triang 30 40)
600
```

4. Calcular el cuadrado de un binomio es la siguiente tarea, que tendrá dos argumentos. De nuevo hay composición de procedimientos. Se trata de

una suma que tiene como argumentos tres productos o dos potencias y un producto, es decir:  $(+ (* a a) (* b b) (* 2 a b))$  o bien  $(+ (expt a 2) (expt b 2) (* 2 a b))$ . Con cajas nos lo podríamos representar así:

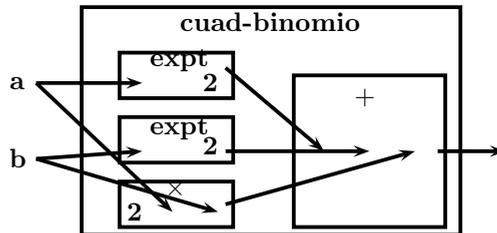


Figura 5.6: Cuadrado de un binomio

Editor

```
#lang scheme
;; texto en el editor

(define cuad-binomio
  (lambda (a b)
    (+ (expt a 2)
       (expt b 2)
       (* 2 a b))
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (cuad-binomio 3 4)
49
```

5. Recuérdese que la tarea consiste en devolver una expresión. Nos serviremos de `printf`, que no devuelve un valor computable.<sup>5</sup>

Editor

```
#lang scheme
;; texto en el editor
(define cuad-binomio
  (lambda (a b)
    (printf
     "(+ (expt ~a 2) (expt ~a 2)
      (* 2 ~a ~a))"
     a b a b)
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (cuad-binomio 3 4)
(+ (expt 3 2) (expt 4 2) (* 2 3
4))
```

<sup>5</sup>El estudiante puede hacer la representación mediante cuadros en papel.



## Unidad 6

# Condicionales

Si un procedimiento se define mediante una expresión *lambda* y el proceso computacional, que la aplicación de un procedimiento produce, depende de la secuencia ordenada de expresiones que conforman el *lambda-body*, el desarrollo de una tarea computacional puede exigir que el proceso consista en la aplicación de ciertas expresiones, y no otras, del *lambda-body*. ¿Cómo programar esa idea? ¿Qué necesitamos? El subconjunto de expresiones del *lambda-body* que han de producir la computación habrá de estar sometido a una *condición* tal que, a la vez que selecciona dicho subconjunto, evita la aplicación del resto de expresiones existentes en el *lambda-body*.

En esta unidad aprenderemos:

- las expresiones condicionales que existen en Scheme,
- y cómo usarlas.

### 6.1. La forma especial `if`

Sea una definición de un procedimiento la siguiente:

```
(define capricho
  (lambda (x)
    (if (>= x 35)
        (set! x (+ x x))
        (set! x (* x x)) )
    x ))
```

En la definición de *capricho* aparece un bloque de texto cuyo primer elemento es `if`. ¿Qué significa dicho bloque?

Expresado en castellano dicho bloque dice lo siguiente: si es verdad que  $X$  es mayor o igual que 35, entonces haz  $X = a$  la suma  $X$  y  $X$ , y si no lo es, haz  $X =$  al producto de  $X$  por  $X$ .

Repárese en que el *lambda-body* está formado por dos expresiones:

1. la expresión `if`:

```
(if (>= x 35)
    (set! x (+ x x))
    (set! x (* x x)) )
```

2. el nombre del único argumento existente:

`x`

`if` es una *forma especial* que introduce una expresión condicional. Abstractamente considerada una expresión `if` sigue el siguiente esquema:

```
(if
  condición
  acción-si-verdadero
  acción-si-falso )
```

donde

1. **condición**: ha de ser una expresión cuya interpretación devuelva un valor booleano;
2. **acción-si-verdadero**: una expresión sintácticamente correcta para Scheme. Será el valor devuelto por `if` si la anterior condición devuelve `#t` (verdadero);
3. **acción-si-falso**: una expresión sintácticamente correcta para Scheme. Será el valor devuelto por `if` si la anterior condición devuelve `#f` (falso).

`if` es una forma especial porque no evalúa todos sus argumentos. En efecto, `if` sólo evalúa la condición y una de las dos posibles acciones.

### 6.1.1. `if` anidados

En algunos casos, nos veremos obligados a usar un `if` en el lugar de alguna de las acciones posibles de otro, es decir como argumento de otro `if`. Veamos un ejemplo:

```
(define otro-capricho
  (lambda (n)
    (if (> n 100)
        (+ n 1)
```

```
(if (> n 50)
    (+ n 2)
    (if (> n 0)
        (+ n 3)
        n) ) )
```

Repárese en que el *lambda-body* está formado por una única expresión.

### 6.1.2. Acciones múltiples con begin

Dentro de un `if` cada una de las acciones posibles ha de consistir en una expresión única, ya sea simple o compuesta. A veces, no obstante, se requiere computar *una secuencia de acciones*. En ese caso, es posible usar la primitiva `begin` en una de las dos siguientes formas:

- acción-si-verdadero/acción-si-falso = `(begin (acción1) (acción2)... (acciónn))` que devuelve el valor de `(acciónn)`.
- acción-si-verdadero/acción-si-falso = `(begin0 (acción1) (acción2)... (acciónn))` que devuelve el valor de `(acción1)`.

Ejemplo:

```
(define capricho
  (lambda (x)
    (if (>= x 35)
        (begin
          (printf "x=~a que es >= que 35~n" x)
          (set! x (+ x x))
        )

        (begin0
          (set! x (* x x))
          (printf "x=~a que es < que 35~n" (sqrt x))
        )
    )
  x ))
```

Bienvenido a DrScheme, versión 4.2.5 [3m].

Lenguaje: scheme.

> (capricho 30)

x=30 que es < que 35

900

> (capricho 36)

x=36 que es >= que 35

72

## 6.2. Las formas when y unless

Cualquiera de estas dos formas de expresión condicional es útil cuando sólo una de las dos acciones posibles de `if` interesa. Si la acción que interesa es cuando la condición es verdadera (**acción-si-verdadero**), se usa **when**. Pero si la acción que interesa es cuando la condición es falsa (**acción-si-falso**), se usa **unless**.

Ejemplos:

```
(define suma-numeros
  (lambda (a b)
    (when (and (number? a) (number? b))
      (display "la suma de los argumentos es ")
      (+ a b) ) ) )
```

```
Bienvenido a DrScheme, versión 4.2.5 [3m].
Lenguaje: scheme.
> (suma-numeros 2 3)
la suma de los argumentos es 5
```

```
(define suma-numeros
  (lambda (a b)
    (unless (and (number? a) (number? b))
      (display "se mostrará un error ")
      (newline) )
    (+ a b) ) )
```

```
Bienvenido a DrScheme, versión 4.2.5 [3m].
Lenguaje: scheme.
> (suma-numeros 2 3)
5
> (suma-numeros 2 'a)
se mostrará un error
+: expects type <number> as 2nd argument, given: a; other arguments
were: 2
```

## 6.3. La forma especial cond

Scheme dispone de otra primitiva, `cond`, que nos permite escribir el texto anterior de una forma más sencilla:

```
(define otro-capricho
  (lambda (n)
    (cond
      ((> n 100) (+ n 1))
      ((> n 50) (+ n 2))
      ((> n 0) (+ n 3))
      (else n) ) ) )
```

`cond` aporta algo que no hay inmediatamente en `if`, a saber: la facilidad de especificar una secuencia de acciones a efectuar si la condición se cumple. Veámoslo modificando un poco nuestra anterior definición:

```
(define otro-capricho
  (lambda (n)
    (cond
      ((> n 100) (display "el argumento dado es: ")
                (display n)
                (display ". El valor devuelto es: ")
                (display (+ n 1)))
      ((> n 50) (display "el argumento dado es: ")
                (display n)
                (display ". El valor devuelto es: ")
                (display (+ n 2)))
      ((> n 0) (display "el argumento dado es: ")
                (display n)
                (display ". El valor devuelto es: ")
                (display (+ n 3)))
      (else (display "el argumento dado es: ")
            (display n)
            (display ". El valor devuelto es: ")
            (display n)) ) ) )
```

Ejemplos de ejecución son:

```
Bienvenido a DrScheme, versión 4.2.5 [3m].
Lenguaje: scheme.
> (otro-capricho 43)
el argumento dado es: 43. El valor devuelto es: 46
> (otro-capricho -65)
el argumento dado es: -65. El valor devuelto es: -65
> (otro-capricho 403)
el argumento dado es: 403. El valor devuelto es: 404
```

`cond` es una forma especial que, al igual que `if`, introduce una expresión condicional. Abstractamente considerada una expresión `cond` sigue el siguiente esquema:

```
(cond
 (condición_1 acción_1 ... acción_n)
 (condición_2 acción_1 ... acción_n)
 ...
 (condición_n acción_1 ... acción_n)
 (else acción_1 ... acción_n) )
```

- Cada expresión «(condición<sub>*i*</sub> acción<sub>1</sub> ... acción<sub>*n*</sub>)» es llamada una cláusula `cond`; donde

- condición: ha de ser una expresión cuya interpretación devuelva un valor booleano;
- acción<sub>1</sub> ... acción<sub>n</sub>: una secuencia de acciones, cada una de las cuales puede ser cualquier expresión sintácticamente correcta para Scheme. Es posible una secuencia vacía. La secuencia se evaluará en el orden en que está escrita. El valor devuelto será el correspondiente a la última acción efectuada: acción<sub>n</sub>;
- (else acción<sub>1</sub> ... acción<sub>n</sub>): caso de que ninguna de las anteriores condiciones se cumpla, el valor de esta cláusula será el valor devuelto por **cond**.

**cond** es una forma especial porque no evalúa todos sus argumentos. En efecto, **cond** evalúa la condición de la primera cláusula y, sólo si dicha condición se satisface, evalúa la secuencia de acciones de esa misma cláusula. Si dicha condición no se cumple, **cond** no evalúa la secuencia de acciones y pasa a la siguiente cláusula, en la que hace lo mismo. Si ninguna condición se cumpliera, **cond** evaluaría la cláusula *else* y devolvería el valor correspondiente a la última acción de esta cláusula.

## Unidad 7

# Procedimientos con condicionales: ejercicios

En esta unidad vamos a hacer ejercicios usando expresiones condicionales. Aprenderemos también a hacer diagramas de flujo, otra técnica auxiliar para la correcta definición de procedimientos. Seguiremos, pues, el siguiente esquema:

1. definir la tarea. Fijando el objetivo y los argumentos;
2. estableceremos los procedimientos existentes con los que hay que contar;<sup>1</sup>
3. escribiremos el código en el editor;
4. y lo probaremos pulsando el botón de ejecutar y aplicando el procedimiento definido;
5. añadiremos el correspondiente diagrama de flujo.

### 7.1. Un uso de *if*

Queremos definir un procedimiento cuya tarea consiste en escribir un mensaje de saludo, si, al pedir un *input*, este es un nombre de persona. Este procedimiento tiene la novedad de que hemos de aprender a solicitar *inputs*.

En Scheme existen varios procedimientos para leer o solicitar *inputs*. Quizá, el más sencillo y general es `read`. `read` permanece solicitando un *input* mientras que no se pulsa . Conviene tener presente varias características de `read`:

- si la expresión introducida es un símbolo, no tiene que estar precedida por `quote` o el apóstrofo;
- si es una lista, tampoco necesita estar precedida por `quote` o el apóstrofo;

---

<sup>1</sup>El estudiante puede seguir usando la técnica de cuadros o bloques de las figuras 5.1 y 5.2

- sin embargo, un `string` ha de escribirse entre comillas.

Por otra parte, el procedimiento que queremos definir requiere contar con una lista de nombres propios conocidos. Podemos usar una *variable* global (la llamaremos «nombres») cuyo valor sea una lista de nombres.

Además, para conservar el *input* introducido, usaremos otra variable (a la que llamaremos «input») cuyo valor será el valor devuelto por `read`.

Por último, la condición que se ha de cumplir es que `input` sea uno de los nombres existentes en `nombres`. Para esto utilizaremos `member`. La sintaxis de `member` es la siguiente: (`member elemento lista`).

Editor

```
#lang scheme

;; un primer uso de if

(define nombres '(Juan juan
Luis luis Mariano mariano Car-
men carmen Pepa pepa Rocío
Rocio rocío rocío))

(define input #f)

(define saludo
  (lambda ()
    (set! input (read))
    (if (member input nombres)
        (printf "Hola ~a" input)
        (printf "Lo siento, no te
conozco"))
    )))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (saludo)
rocio eof
Hola rocío
> (saludo)
Rocío eof
Hola Rocío
> (saludo)
Antonio eof
Lo siento, no te conozco
>
```

Esta idea puede ser representada mediante el siguiente diagrama de flujo (figura 7.1):

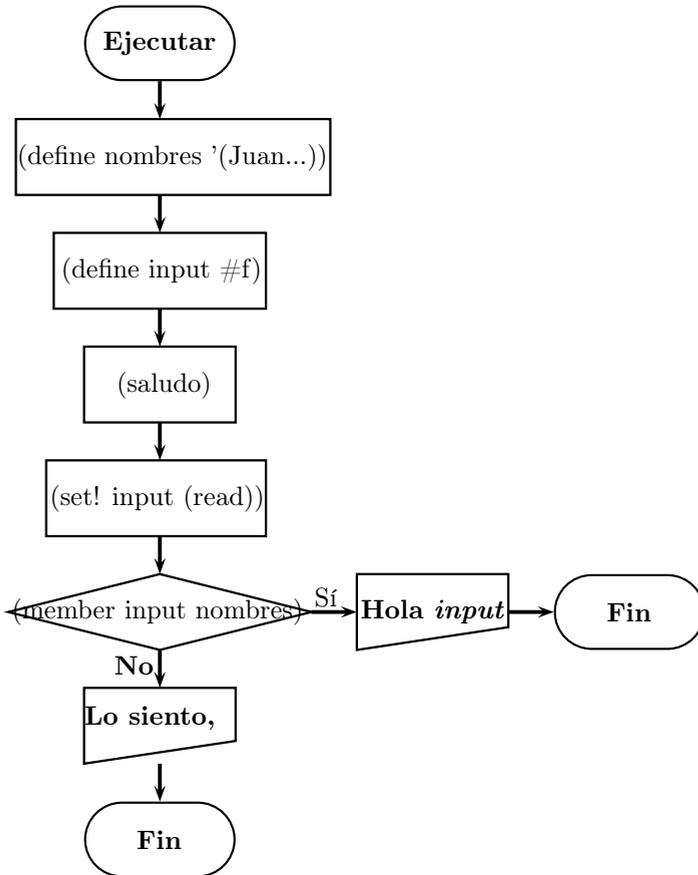


Figura 7.1: Programa de saludo

## 7.2. Diagramas de flujo

Los diagramas de flujo usan convencionalmente ciertas figuras que conviene conocer:

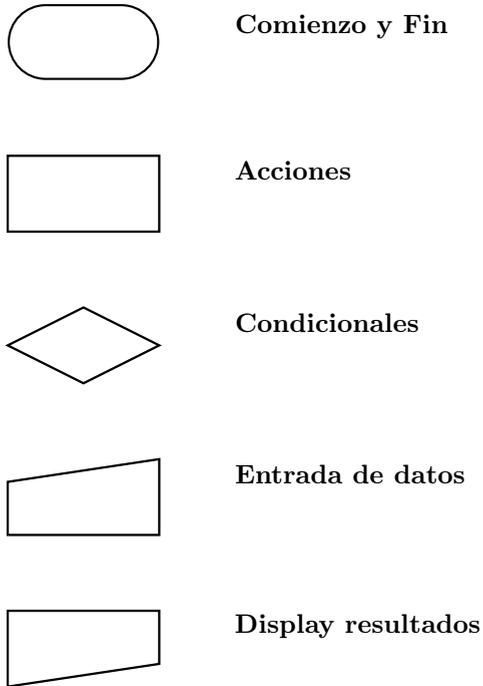


Figura 7.2: Figuras en diagramas de flujo

## 7.3. Uso de *ifs* anidados

Si queremos definir un procedimiento que discrimine, en primer lugar, entre números y el resto de datos, y que, en caso de que el argumento sea un número, discrimine entre enteros y cualquier otro tipo de número, y, finalmente, discrimine entre enteros positivos y negativos, podemos usar varios *ifs* anidados.

Podemos precisar esta idea así:

1. Primer *if*:

```
(if (number? num)
    ...
    (printf "El argumento ~a no es un número" num) )
```

2. Anidamos un segundo if:

```
(if (number? num)
    (if (integer? num)
        ...
        (printf "El número ~a no es un entero" num) )
    (printf "El argumento ~a no es un número" num) )
```

3. Finalmente, añadimos un tercer if:

```
(if (number? num)
    (if (integer? num)
        (if (positive? num)
            (printf "~a es un entero positivo" num)
            (printf "~a es un entero negativo" num))
        (printf "El número ~a no es un entero" num) )
    (printf "El argumento ~a no es un número" num) )
```

Editor

Intérprete

```
#lang scheme

;; ifs anidados

(define tipo-de-entero
  (lambda (num)
    (if (number? num)
        (if (integer? num)
            (if (positive? num)
                (printf "~a es un entero
positivo" num)
                (printf "~a es un entero
negativo" num))
            (printf "El número ~a no es
un entero" num) )
        (printf "El argumento ~a no es
un número" num) )
    ))
```

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (tipo-de-entero 23)
23 es un entero positivo
> (tipo-de-entero -23)
-23 es un entero negativo
> (tipo-de-entero 0.23)
El número 0.23 no es un entero
> (tipo-de-entero 23bis)
El argumento 23bis no es un
número >
```

## 7.4. Un uso de *cond*

El ejercicio consiste en definir un procedimiento de un único argumento cuyo fin es identificar el tipo de dato al que pertenece el argumento dado. Según que el argumento dado pertenezca a un tipo u otro, se escribirá en pantalla una determinada respuesta. Los procedimientos requeridos son: `cond`, `printf` y algunos

predicados de tipo. `cond` permite introducir varias condiciones distintas en una secuencia de cláusulas. `printf` será la acción correspondiente en cada cláusula, si bien con mensajes adecuados a la condición en cada caso.

Editor

```
#lang scheme
;; texto en el editor

(define dato-del-tipo
  (lambda (arg)
    (cond
      ((number? arg)
       (printf "el argumento es un
número"))
      ((char? arg)
       (printf "el argumento es un
carácter"))
      ((symbol? arg)
       (printf "el argumento es un
símbolo"))
      ((pair? arg)
       (printf "el argumento es un
par"))
      ((list? arg)
       (printf "el argumento es una
lista"))
      ((string? arg)
       (printf "el argumento es un
string"))
      ((vector? arg)
       (printf "el argumento es un
vector"))
      (else
       (printf 'desconocido)))
  )))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (dato-del-tipo 34)
el argumento es un número
> (dato-del-tipo #'w)
el argumento es un carácter
> (dato-del-tipo 'ferrocarril)
el argumento es un símbolo
> (dato-del-tipo (cons 1 q))
el argumento es un par
> (dato-del-tipo (list 1 (list 2 3)
't))
el argumento es un par
> (dato-del-tipo "sevilla")
el argumento es un string
> (dato-del-tipo (vector "león"
123))
el argumento es un vector
> (dato-del-tipo #f)
desconocido
>
```

Obsérvese el caso de `(dato-del-tipo (list 1 (list 2 3) 't))`. Esperamos *lista*, como respuesta, y, sin embargo, la respuesta es: *par*. ¿Por qué? Porque toda lista (propia o impropia) es un par, pero una lista impropia (un par) no es una lista. El problema puede resolverse alterando el orden de la secuencia de cláusulas, anteponiendo `((list? arg) (printf "el argumento es una lista"))` a `((pair? arg) (printf "el argumento es un par"))`. Es lo que vimos en el apartado 4.2.1. Esta idea puede ser representada mediante el siguiente diagrama de flujo (figura 7.3):

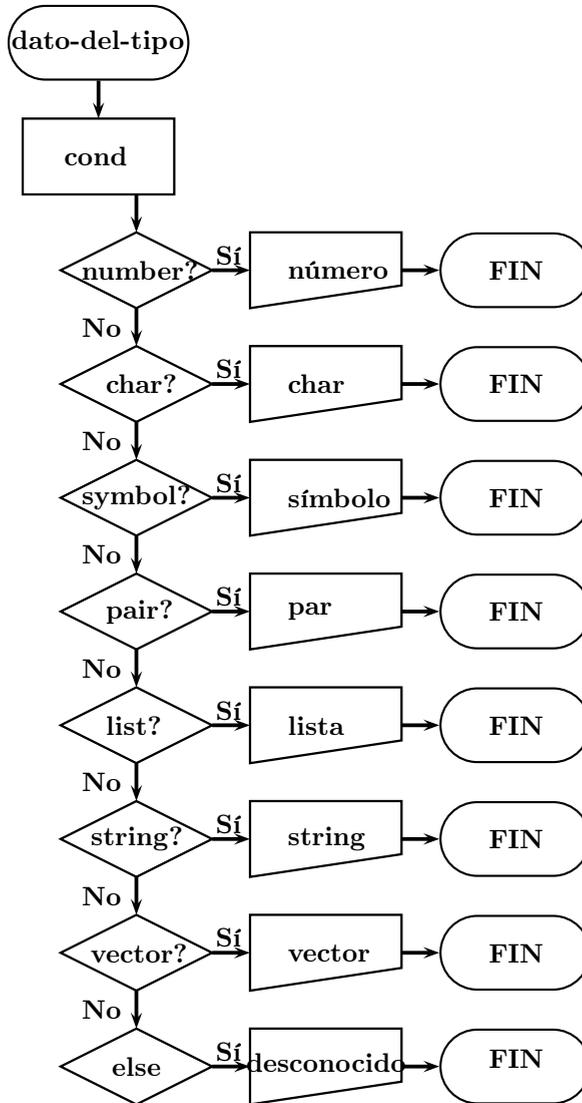


Figura 7.3: dato-del-tipo

## 7.5. Uso de *when* y *unless*

**when** es un condicional que tiene una única acción, cuando la condición es *verdadera*. La acción que realiza **when** puede ser una secuencia de expresiones, en este caso, **when** devuelve el valor de la última expresión.

Definamos un procedimiento de dos argumentos, cada uno ha de ser un símbolo. La condición que ha de cumplirse es que sean iguales.

Editor

```
#lang scheme

;; when

(define simbolos-iguales
  (lambda (a b)
    (when (eq? a b)
      (printf "~a = ~a~n" a b)
    )
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (simbolos-iguales 'dedo 'dedo)
dedo = dedo
> (simbolos-iguales 'dedo 'Dedo)
>
```

**unless** es otro condicional que también tiene una única acción, cuando la condición es *falsa*. También la acción que realiza **unless** puede ser una secuencia de expresiones, en este caso, **unless** devuelve el valor de la última expresión.

Ahora definimos un procedimiento de dos argumentos, cada uno ha de ser un símbolo. La condición que ha de cumplirse es que no sean iguales.<sup>2</sup>

Editor

```
#lang scheme

;; unless

(define simbolos-desiguales
  (lambda (a b)
    (unless (eq? a b)
      (printf "~a no es igual que
~a~n" a b)
      (printf "~a~n" a)
      (printf "~a~n" b)
    )
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (simbolos-desiguales 'tres
'tres)
> (simbolos-desiguales 'tres
'cuatro)
tres no es igual que cuatro
tres
cuatro
>
```

<sup>2</sup>Se aconseja al estudiante que haga los diagramas de flujo de ambos procedimientos con papel y lápiz.

## Unidad 8

# Variables locales

En el apartado 7.1 hemos visto un pequeño programa en el que se usan dos *variables globales* y se define el procedimiento `saludo`. Si se reflexiona un poco, se advertirá que la segunda variable global (`input`) sólo se utiliza dentro de la aplicación del procedimiento `saludo`. ¿No podríamos incorporar su definición al texto de la definición del procedimiento `saludo`? En efecto, es posible. Sin embargo, al hacerlo, la variable `input` se convierte en una *variable local*, porque cambia su *accesibilidad*: en este caso, `input` sólo puede ser usada en el contexto establecido por el texto de la definición de `saludo`.

Para entenderlo mejor, estudiemos una modificación del programa mencionado. Recordemos:

Editor

```
#lang scheme
;; un primer uso de if
(define nombres '(Juan juan Luis
luis Mariano mariano Carmen
carmen Pepa pepa Rocío Rocio
rocío rocío))

(define input #f)
(define saludo
  (lambda ()
    (set! input (read))
    (if (member input nombres)
        (printf "Hola ~a" input)
        (printf "Lo siento, no te
conozco")) )
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> input
#f > (saludo)
Luis eof
Hola Luis
> input
Luis >
```

Obsérvese que `input` es accesible antes de aplicar y después de aplicar el procedimiento `saludo`. Ahora, lo que proponemos hacer es lo siguiente:

Editor	Intérprete
<pre>#lang scheme ;;variables locales  (define nombres '(Juan juan Luis luis Mariano mariano Car- men carmen Pepa pepa Rocío Rocio rocío rocío))  (define saludo   (lambda ()     (define input (read))     (if (member input nombres)         (printf "Hola ~a" input)         (printf "Lo siento, no te co- nozco" )     ))   ))</pre>	<pre>Bienvenido a DrScheme, versión 4.2.5 [3m]. Lenguaje: scheme. &gt; input . . . reference to an identifier before its definition: input &gt;(saludo) Pepa eof Hola Pepa &gt; input . . . reference to an identifier before its definition: input &gt;</pre>

Como se ve, `input` ya no es una variable global y, por ello, en el entorno global del intérprete deja de ser accesible. Sin embargo, en el proceso computacional resultante de aplicar `saludo` sí es accesible, pues de no serlo en dicho proceso también se produciría un error.

En esta unidad aprenderemos:

1. las distintas formas de las variables locales;
2. cómo usarlas

## 8.1. Formas de las variables locales

Ya hemos visto que la forma `(define variable valor)` puede usarse dentro del texto de la definición de un procedimiento. Esa forma de variable local permite también el uso de `set!` para cambiar el valor de la variable definida localmente.

No obstante, la forma más usual de crear variables locales en la definición de un procedimiento consiste en usar la forma especial `let` y sus variantes. El esquema general de `let` es la siguiente:

```
(let (
  (variable-local-1 valor-1)
  (variable-local-2 valor-2)
  ...
```

```

    (variable-local-n valor-n)
  )
<let-body>
)

```

Un ejemplo es una modificación del anterior ejemplo de programa:

Editor

```

#lang scheme
;;variables locales

(let (
  (nombres '(Juan juan Luis
luis Mariano mariano Carmen
carmen Pepa pepa Rocío Rocío
rocío rocío))
  (input #f)
  )
  (set! input (read))
  (if (member input nombres)
      (printf "Hola ~a" input)
      (printf "Lo siento, no te
conozco")))
  )

```

Intérprete

```

Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
Juan eof
Hola Juan
> nombres
. . reference to an identifier
before its definition: nombres
>input
. . reference to an identifier
before its definition: input
>

```

### 8.1.1. Computación en el *let-body*

En el ejemplo anterior el *let-body* está formado por una secuencia de expresiones que producen un proceso computacional. Si se compara el texto de este ejemplo con el texto de la definición de saludo en el primer ejemplo (página 61), se verá que el *let-body* coincide con el texto del *lambda-body*. Las conclusiones que hemos de sacar son, por ello, las siguientes:

1. el bloque de texto de un `let` permite realizar una computación. Esta computación se define mediante una secuencia finita de expresiones en el *let-body*;
2. las variables locales definidas en el texto de `let` sólo son accesibles para las expresiones que conforman el *let-body*;
3. al ejecutar un bloque de texto `let`, ninguna de las variables, localmente definidas en el texto de `let`, es accesible más allá de su computación;
4. el valor de las variables locales de un texto `let` sólo puede ser modificado, durante el proceso computacional, con `set!`.

### 8.1.2. Variantes de *let*: las variables locales

Los *valores* de las variables locales de un bloque de texto **let** no pueden determinarse *mencionando el nombre* de una variable-let previamente definida. Sin embargo, sí es posible hacer eso en otras formas de **let**. Otras formas de **let** son:

- **let**. Esta es la primera variante de **let**, la que acabamos de estudiar;
- **let\***. Con esta variante es posible mencionar el nombre de una *variable-let\** previamente establecida. Recuérdese que mencionar el nombre de una variable obliga a Scheme a poner en su lugar el valor que tenga asociado, cuando la computación se produce.
- **letrec**. La estudiaremos en la unidad de título «recursión»;
- **let etiqueta**. La estudiaremos en la unidad de título «recursión».

## 8.2. Accesibilidad de variables

Cuando cargamos DrScheme, en la ventana del intérprete, estamos en el *entorno global*. En dicho entorno existen todos los procedimientos primitivos que el intérprete contiene: p. ejemplo, **+**, **\***, **/**, **-**, **define**, **quote**, etc. El conjunto de los procedimientos primitivos es un conjunto de variables globales. Cada uno de estos procedimientos primitivos es accesible para todos los demás. Ya lo hemos visto: se puede escribir una expresión en la que un procedimiento primitivo sea argumento de otro procedimiento primitivo. La definición que hagamos de cualquier variable global o de cualquier procedimiento se inscribe y suma al entorno global.

De manera que es posible la siguiente interacción con el intérprete:

```
(define global 73)

(define añade1
  (lambda (n . x)
    (add1
     (apply + (cons n x)))))
```

Al ser aplicado el procedimiento *añade1* con la variable *global* como uno de sus argumentos da la siguiente respuesta:

```
Bienvenido a DrScheme, versión 4.2.5 [3m].
Lenguaje: scheme.
> (añade1 2 4 global)
80
```

La expresión `(añade1 2 4 global)`, hace uso de la variable definida en la primera línea: `(define global 73)`, del texto del programa. Es decir, se trata de una

variable que es accesible desde cualquier punto del texto del programa; por ello, se dice que su alcance léxico es global. Una ligera modificación del código del ejemplo lo mostrará mejor:

```
(define global 73)

(define añade1
  (lambda (n . x)
    (set! global 37)
    (add1
     (apply + (cons n x))) ))
```

```
Bienvenido a DrScheme, versión 4.2.5 [3m].
Lenguaje: scheme.
> (añade1 2 4 global)
80
> (añade1 2 4 global)
44
```

La línea `(set! global 37)` no es procesada hasta que no se produce una aplicación de `añade1`, cosa que sucede con la primera línea en que se ha escrito `(añade1 2 4 global)`. La segunda aplicación ya hace uso del nuevo valor asignado a `global`.

### La primitiva *apply*

El procedimiento primitivo `apply` tiene la siguiente sintaxis:

```
(apply procedimiento argumento)
```

`apply` es una forma especial que transforma la expresión que la contiene según el siguiente esquema:

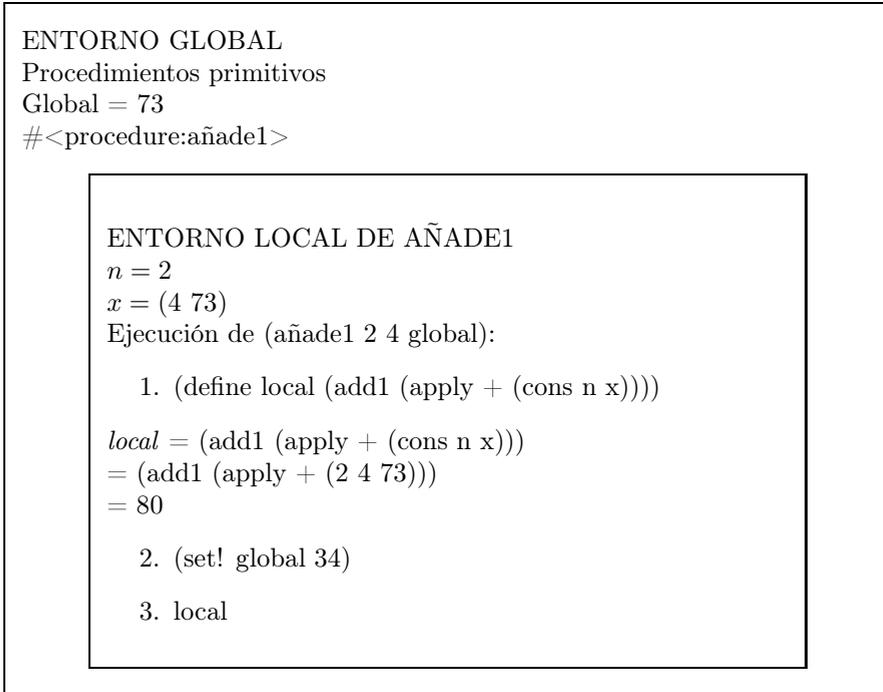
```
(apply procedimiento argumento) ⇒ (procedimiento argumentos)
```

Por ejemplo, `(apply + (list 1 2 3 4))` produce, primero, la expresión `(+ 1 2 3 4)` y la aplica. Otros ejemplos:

- `(apply * (list 2 3)) ⇒ (* 2 3) ⇒ 6`
- `(apply cons (list 'a (list 2 3))) ⇒ (cons 'a (2 3)) ⇒ (a 2 3)`
- `(apply append (list (list 'a 'b) (list 2 3))) ⇒ (append (a b) (2 3)) ⇒ (a b 2 3)`

### 8.2.1. Entornos locales como entornos anidados

Una forma intuitiva de pensar la relación entre el entorno global y un entorno local es la siguiente:



La relación entre ambos entornos es asimétrica: mientras que desde el entorno local de `añade1` es posible usar cualquier variable definida en el entorno global, lo contrario no es posible. Esta idea se refuerza con la siguiente consideración: es posible que dos variables tengan el mismo identificador si, y sólo si, están en distintos entornos. ¿Por qué? Porque las reglas de evaluación vistas en la Unidad 2 se subordinan a los entornos. Lo cual implica que: 1.º, se aplican en el entorno actual de interpretación y si fallan en éste, 2.º, se ensayan en el entorno que contiene al anterior; 3.º, sólo si se produce un error en el entorno final o global, se devuelve un mensaje de error.

Por último, hay que tener en cuenta que los argumentos de los procedimientos son variables de pleno derecho, cuando el procedimiento es aplicado. ¿Por qué? Porque sólo en ese momento responde a la definición que hemos dado de variable:

$$\text{Variable} = \langle \text{nombre}, \text{valor} \rangle$$

En efecto, sólo al ejecutar un procedimiento se produce la vinculación entre un argumento (identificador) y un valor. Pero, como acabamos de ver, dicha vinculación se produce en el entorno local creado al computar el procedimiento con argumentos concretos. Esto mismo sucede con cualquier otra variable definida *dentro* de la definición del procedimiento en cuestión.

## Unidad 9

# Recursión

En un libro de Dave Touretzky<sup>1</sup> se cuenta una historia bonita para introducir la idea de procesos recursivos. Es, en síntesis, la siguiente:

A un joven aprendiz de unos 15 años se le encomendó una tarea: averiguar si en un saco lleno de bolas había alguna que tuviera escrito un número impar. El saco era especial, porque sólo permitía introducir la mano y coger una bola y sacarla. El joven no sabía qué hacer. En el castillo estaba encerrado en una mazmorra un dragón, ya viejo y malhumorado por los años de cautiverio. El joven sabía que el dragón era el único habitante inteligente del castillo. Y al dragón se dirigió con su cuita. Al principio, el dragón, sin llegar a echar fuego, se mostró desabrido y poco o nada dispuesto a colaborar. Después de varios refunfuños, el dragón miró fijamente al joven y dijo:

— ¿Eres tonto? Haz lo único que puedes hacer.

El joven, que aún no se había recuperado de la impresión que le producía el dragón, no entendió y volvió a pedir al dragón que le dijera lo que tenía que hacer. El dragón volvió a mirarle, movió la cabeza, y le espetó:

— Efectivamente eres tonto. ¡Que hagas lo único que puedes hacer!

Aquello surtió efecto y el joven empezó a pensar qué quería decir el dragón. Al rato, empezó a hablar en voz alta:

— Lo único que puedo hacer es meter la mano en el saco y sacar una bola.

Calló. Al rato volvió a hablar:

— Si la bola que saque tiene escrito un número impar, he resuelto el problema. Y si no. . .

Mientras el joven pensaba, el dragón dijo:

— Pues tira la bola y vuelve a empezar.

Al oírlo, el joven sonrió y su rostro adquirió el resplandor de la felicidad. Volvió a hablar:

— Quieres decir que repitiendo la acción de sacar una bola, comprobando si es impar su número, dejarla si no lo es, puedo saber la respuesta. ¡Claro, que tonto

---

<sup>1</sup> *Common Lisp: A Gentle Introduction to Symbolic Computation*. Addison-Wesley / Benjamin Cummings, 1990.

soy! Si alguna bola tiene escrito un impar, el problema está resuelto y, si no, cuando el saco esté vacío también tendré la respuesta: no hay un impar. ¡Perfecto!, eres un genio.

El dragón, aunque parezca mentira, se sonrió. El joven volvió apresuradamente a sus aposentos.

## 9.1. Componentes de los procesos recursivos

El cuento anterior nos puede servir para entresacar los elementos esenciales de un proceso recursivo. Podemos sintetizarlos así:

- hay una acción que se repite una y otra vez;
- hay una condición que, de cumplirse, evita el tener que continuar con la acción: nos proporciona una respuesta;
- hay otra condición que exige de continuar con la acción indefinidamente y que también nos proporciona una respuesta.

Mientras que ambas condiciones se pueden expresar mediante una frase fija e inmutable, la acción que se repite una y otra vez, no, porque cambia. Si nos fijamos en el cuento, la acción consiste en sacar una bola del saco cada vez. Lo que cambia es el saco, pues de una a otra vez contendrá menos bolas, exactamente una menos cada vez. ¿Qué exige la acción de meter la mano y sacar una bola? El saco, naturalmente. Si etiquetamos la acción de meter la mano y sacar una bola con el nombre «saca-bola», **saca-bola** puede ser el nombre de un procedimiento cuyo único argumento es «saco». De manera que el proceso recursivo puede establecerse así:

1. inicio: (**saca-bola** **saco**)
2. si es verdad (**impar?** **la-bola-sacada**)  $\Rightarrow$  la respuesta es: sí, y aquí concluye el proceso.
3. si no, (**saca-bola** (**resto-del** **saco**))
4. cuando el saco esté vacío, (**vacío** **saco**), la respuesta es: no, y concluye el proceso.

Probemos a definir un procedimiento para esas ideas. Lo llamaremos «hay-un impar?». Su argumento puede ser representado mediante una lista de números. Así, lo que llamábamos *saco*, puede ser llamado «lista». La acción consistirá en sacar el primero de la lista (**car** o **first**) y comprobar si es un impar. Si lo es, se devuelve **#t**. Y si no lo es, se vuelve a llamar o ejecutar **hay-un impar?**, pero con el resto de la lista (**cdr** o **rest**). Cuando la lista esté vacía (**null?**), se devuelve **#f**. Llamaremos a esta última condición «condición de parada» o «caso mínimo». La condición de parada suele ser la primera que se comprueba en los procedimientos recursivos.

Editor

```
#lang scheme
;; recursión

(define hay-un-impar?
  (lambda (lista)
    (cond
      ((null? lista) #f)
      ((odd? (first lista)) #t)
      (else (hay-un-impar? (rest
lista))))
  ) )
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (hay-un-impar? '(2 4 6 8 10
12))
#f
> (hay-un-impar? '(2 4 6 8 10 11
12))
#t
> (hay-un-impar? '(1 2 4 6 8 10
12))
#t
>
```

### 9.1.1. El rastro de hay-un-impar?

DrScheme tiene una utilidad que nos permite rastrear lo que sucede en un proceso computacional. Para ello hay que incluir en el editor la línea siguiente: `(require mzlib/trace)` y añadir también `(trace nombre-procedimiento)`. Si las incluimos en el texto anterior, veremos lo siguiente:

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define hay-un-impar?
  (lambda (lista)
    (cond
      ((null? lista) #f)
      ((odd? (first lista)) #t)
      (else (hay-un-impar? (rest
lista))))
  ) )
(trace hay-un-impar?)
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (hay-un-impar? '(2 4 6 8 10
12))
> (hay-un-impar? (2 4 6 8 10
12))
> (hay-un-impar? (4 6 8 10 12))
> (hay-un-impar? (6 8 10 12))
> (hay-un-impar? (8 10 12))
> (hay-un-impar? (10 12))
> (hay-un-impar? (12))
> (hay-un-impar? ())
< #f
#f >
```

### 9.1.2. Ejercicios explicados

Se trata de familiarizarse con la definición de procedimientos recursivos. Vamos a definir y explicar los siguientes procedimientos:

1. Carcajadas. Un procedimiento con un argumento: un número natural. No devuelve un valor computable, sino que muestra en pantalla tantos «ja» como indique su argumento.
2. Intervalo. Un procedimiento con dos argumentos: el primero, un número con el que comienza una serie; el segundo, otro número en el que acaba la serie. El valor devuelto será una lista integrada por dicha serie de números.
3. Factorial. Un procedimiento con un argumento: un número natural. Calcula y devuelve el factorial de dicho número según la siguiente definición: factorial de  $n = n \times \text{factorial}(n - 1)$ .

### Carcajadas

La acción que hay que realizar, cada vez, es escribir en pantalla «ja». Para ello, podemos usar `printf` o `display`. Y como hay que escribirlo tantas veces como indique el número, cada vez que escribamos «ja» haremos, a continuación, una llamada recursiva restando 1 al número dado. La condición de parada será que dicho número sea 0.

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define carcajadas
  (lambda (n)
    (if (> n 0)
        (begin
          (display 'ja)
          ;(printf "ja")
          (carcajadas (- n 1)))
        (newline) )
    ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (carcajadas 10)
ja.ja.ja.ja.ja.ja.ja.ja.ja
>
```

Obsérvese que la definición establece la condición de parada de forma tácita, porque la acción recursiva se lleva a cabo mientras que el número sea mayor que 0.

### Intervalo

El problema consiste, en este procedimiento, en construir una lista de números a partir de un número dado hasta otro número también dado. Por tanto, la acción

repetida será introducir el primer número en la lista de los restantes. Supongamos que los números fueran 1 y 2. Lo que hay que conseguir es la lista (1 2). Pero, una vez que se ha introducido el primer número en la lista, dicho número ha de cambiar al siguiente. La idea es la siguiente:

1. A (intervalo 1 2) ha de seguir:

a) (cons 1...

b) (intervalo (+ 1 1) 2).

¿Cuál es la condición de parada? ¿Qué hay que devolver cuando se cumpla? La condición de parada no puede ser que ambos números sean iguales, sino que el primero sea mayor que el segundo. En este caso, como el procedimiento ha de devolver una lista, se devolverá una lista vacía. En efecto, si se ejecuta (intervalo 2 2), ¿qué debe devolver la ejecución? Naturalmente, la lista (2), porque 2 es el único número que hay en esa serie. Pero si se aplica (intervalo 3 2), ¿qué debe devolver la ejecución? La lista vacía, porque no hay ningún número mayor o igual que 3 y menor que 2. Por tanto, la condición de parada puede expresarse así (llamando «a» y «b» a los argumentos):  $(> a b) \Rightarrow '()$ .

Resumiendo tenemos lo siguiente:

1. es un procedimiento con dos argumentos;
2. en cada llamada recursiva del procedimiento, se incrementa el primer argumento (número inferior de la serie) en 1;
3. la acción repetida consiste en introducir el primer argumento en la lista que se irá produciendo con las llamadas recursivas;
4. la condición de parada es que el primer argumento sea mayor que el número en que ha de finalizar la serie (segundo argumento).

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define intervalo
  (lambda (a b)
    (if (>a b)
        '()
        (cons a
              (intervalo (+ 1 a) b))))))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (intervalo 1 10)
(1 2 3 4 5 6 7 8 9 10)
> (intervalo 5 5) (5)
> (intervalo 11 10)
()
```

Al rastrear `(intervalo 1 10)` se produce lo siguiente:

```
> (intervalo 1 10)
>(intervalo 1 10)
> (intervalo 2 10)
> >(intervalo 3 10)
> > (intervalo 4 10)
> > >(intervalo 5 10)
> > > (intervalo 6 10)
> > > >(intervalo 7 10)
> > > > (intervalo 8 10)
> > > > >(intervalo 9 10)
> > > > > (intervalo 10 10)
> > > > > [10] (intervalo 11 10)
-----
< < < <[10] ()
< < < < < (10)
< < < < <(9 10)
< < < < (8 9 10)
< < < <(7 8 9 10)
< < < (6 7 8 9 10)
< < <(5 6 7 8 9 10)
< < (4 5 6 7 8 9 10)
< <(3 4 5 6 7 8 9 10)
< (2 3 4 5 6 7 8 9 10)
<(1 2 3 4 5 6 7 8 9 10)
(1 2 3 4 5 6 7 8 9 10)
```

Obsérvese que,

1. en cada nueva llamada recursiva, el primer argumento se incrementa en 1;
2. la anterior llamada del procedimiento no termina de ejecutarse y queda aplazada;
3. cuando la condición de parada se cumple, la última llamada recursiva devuelve un valor: la lista vacía;
4. ese valor es pasado a la penúltima llamada recursiva, cuya acción aplazada fue `(cons 10 (intervalo (+ 1 10) 10))`;
5. lo que produce la expresión siguiente: `(cons 10 '())`  $\Rightarrow$  `(10)`;
6. el proceso continúa realizando ordenadamente cada una de las aplicaciones del procedimiento aplazadas.

## Factorial

La definición del factorial de un número natural es: factorial de  $n = n \times \text{factorial}(n - 1)$ . Intentando reproducir esta definición en Scheme, se tienen las siguientes expresiones:

```
(* n
  (factorial (- n 1)))
```

La condición de parada será que  $n = 1$ , puesto que el 1 es el primer número de la serie de los naturales. Además la operación se aplica a cualquier número natural y devuelve siempre otro número natural o, dicho de otra manera, es una operación que parte del conjunto de los números naturales y acaba en ese mismo conjunto. Lo cual obliga a introducir explícitamente otra condición que nos asegure que la operación se lleva a cabo en el conjunto de los naturales, a saber: `(and (integer? n) (positive? n))`

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define factorial
  (lambda (n)
    (if (and (integer? n) (positive? n))
        (if (= n 1)
            1
            (* n
              (factorial (- n 1)) )
            (printf -a no es un número
                    natural"n) )
        ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (factorial 0)
0 no es un número natural
> (factorial -2)
-2 no es un número natural
> (factorial 4)
24
> (factorial 5)
120
>
```

Al observar el rastro de factorial, se comprueba que también hay una primera fase caracterizada por operaciones aplazadas y una segunda fase de realización de cada operación aplazada:

```
> (factorial 5)
>(factorial 5)
> (factorial 4)
>>(factorial 3)
>> (factorial 2)
>>>(factorial 1)
```

-----  
< < <1

```
< < 2
< <6
< 24
<120
120
```

### 9.1.3. Ejercicios resueltos

Para continuar el aprendizaje sobre procedimientos recursivos, vamos a hacer varios ejercicios más. Ahora se explicarán las especificaciones de cada procedimiento y, a continuación, se dará el código correspondiente a cada uno.

Primer ejercicio:

1. Tarea del procedimiento: formar una lista a partir de un número dado y una lista de números dada, haciendo que cada elemento de la lista resultante sea la suma de cada número de la lista y el número dado.
2. Daremos al procedimiento el nombre `suma-numero-lista` y a los argumentos `numero` y `lista`, respectivamente.
3. Hay dos operaciones que conviene separar:
  - a) para sumar el número dado a cada elemento de la lista inicial, hay que recorrer, elemento a elemento, la lista. Por ello y supuesto que cada vez extraemos el primero de la lista, en cada llamada recursiva la lista ha de menguar, es decir ha de ser la lista menos el primero (`rest lista`).
  - b) el valor que el procedimiento ha de devolver es otra lista (en la que cada elemento es el valor de la suma realizada), lista que hay, por tanto, que construir paso a paso.
4. La condición de parada será que la lista dada como argumento esté vacía.

El código que plasma estas ideas es:

```
#lang scheme
(require mzlib/trace)

;; recursión

(define suma-numero-lista
  (lambda (numero lista)
    (if (null? lista)
        '()
        (cons
         (+ numero (first lista))
         (suma-numero-lista numero (rest lista)))) )
  ))
```

### Operaciones con listas

El anterior procedimiento puede llevarse a cabo mediante el procedimiento primitivo `map`, que vamos a estudiar ahora. La sintaxis de `map` es:

```
(map procedimiento lista)
```

`map` tiene las siguientes características:

1. El valor que devuelve es una lista.
2. La acción consiste en computar el procedimiento tantas veces cuantos elementos tenga la lista.
3. Al computar el procedimiento, su argumento tendrá como valor el primer elemento no usado aún de la lista.

Veamos el anterior ejemplo en un código que usa `map`:

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define suma-numero-lista
  (lambda (numero lista)
    (map
     (lambda (elemento)
       (printf "primer elemento de
lista = ~a~n" elemento)
       (+ numero elemento)))
     lista)
  ))
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (suma-numero-lista 1 '(1 2 3))
primer elemento de lista = 1
primer elemento de lista = 2
primer elemento de lista = 3
(2 3 4)
>
```

### Expresiones *lambda* anónimas

Obsérvese, por último, el uso de *lambda*. Hay que recordar que una *expresión lambda* es, normalmente, el valor de un identificador: el nombre del procedimiento. Pero si la *expresión lambda* es el valor de un nombre, ha de ser un objeto que puede usarse de pleno derecho. ¿Cómo se usa una *expresión lambda*? Basta recordar las reglas básicas de Scheme respecto a los nombres y cómo se aplica un procedimiento. Por ejemplo, una expresión correcta para aplicar `suma-numero-lista` es `(suma-numero-lista 1 '(1 2 3))`. Al interpretar esa expresión, Scheme sustituye `suma-numero-lista` por su valor, es decir, por `(lambda (numero lista)` .... Con ello se tiene otra expresión igualmente correcta, a saber:

```
( (lambda (numero lista)
  (map
   (lambda (elemento)
    (printf "primer elemento de lista = ~a~n" elemento)
    (+ numero elemento))
   lista)
  ) 1 '(123) )
```

que es la que Scheme evalúa. Consecuencia de esto es que es posible usar *expresiones lambda* sin que estén vinculadas a un nombre. Es lo que se conoce como el uso *anónimo de lambda*.

Segundo ejercicio:

1. Tarea: calcular la media aritmética de una lista de números. Es decir que hay que dividir la suma de todos los números por la cantidad de números que haya en la lista.
2. Es posible, y probablemente más claro, definir dos procedimientos que hagan la tarea.
3. Daremos el nombre de **sumatorio** al que sirva para obtener la suma de todos los números presentes en la lista. Y **media-arit** al que calcule la media aritmética.
4. **sumatorio** será un procedimiento recursivo que:
  - a) recorre la lista de números, del primero al último. Su condición de parada será que la lista esté vacía.
  - b) aplazándola, en cada llamada recursiva suma el primero de la lista al valor devuelto por la siguiente aplicación. Cuando la condición de parada se cumpla y la lista esté vacía, se devolverá 0.
5. **media-arit** es un procedimiento no recursivo que, no obstante, también hace dos sub-tareas:
  - a) dividir el valor aportado por **sumatorio** por el número total de elementos de la lista.
  - b) y dado que la operación de calcular la media aritmética ha de hacerse sobre una lista de números, hay que excluir la posibilidad de que dicha lista sea la lista vacía. En otras palabras, el conjunto de partida está formado por listas finitas de números con exclusión de la lista vacía.

El programa tiene, pues, dos definiciones:

```
(define sumatorio
  (lambda (lista)
    (if (null? lista)
```

```

    0
    (+ (first lista)
       (sumatorio(rest lista))) )
  ))

(define media-arit
  (lambda (lista)
    (if (null? lista)
        (printf "la lista vacía no forma parte del conjunto
de las listas de números")
        (/ (sumatorio lista) (length lista)) )
    ))

```

Tercer ejercicio:

1. Tarea del procedimiento: buscar en una lista de números el mayor de todos ellos.
2. Separaremos la tarea en dos procedimientos. Uno será recursivo, mientras que el otro excluirá la posibilidad de la lista vacía, porque esta no forma parte del conjunto formado por listas finitas de números.
3. Procedimiento recursivo:
  - a) tendrá dos argumentos: **el-mayor** y **sublista**.
  - b) la recursión consistirá en recorrer la lista hasta que esté vacía. Esta será la condición de parada.
  - c) en cada aplicación se comprobará si **el-mayor**, inicialmente el primero de la lista, es efectivamente mayor que el primero de **sublista**.
  - d) ¿por qué? Basta con pensar en el caso de una lista como: '(3). El primero se coloca como supuesto mayor: **el-mayor** = 3, la lista se hace igual al resto: '(). Al cumplirse la condición de parada, se devuelve el valor de **el-mayor**.
  - e) pero si resulta que **el-mayor** no es efectivamente mayor que el primero de **sublista**, se hace de este **el-mayor** y se continua recorriendo el resto de la lista.
4. Procedimiento no recursivo:
  - a) excluye la posibilidad de la lista vacía.
  - b) aplica el procedimiento recursivo tomando como **el-mayor** al primero de la lista y como **sublista** al resto de la lista.

## Editor

```

#lang scheme
(require mzlib/trace)
;; recursión

(define mayor
  (lambda (lista)
    (if (null? lista)
        (printf "la lista vacía no
forma parte del conjunto de las
listas de números")
        (mayor-rec (first lista) (rest
lista)) )
    ))

(define mayor-rec
  (lambda (el-mayor sublista)
    (if (null? sublista)
        el-mayor
        (if (>(first sublista) el-
mayor)
            (mayor-rec (first sublista)
(rest sublista))
            (mayor-rec el-mayor (rest
sublista)) )
        )
    ))
(trace mayor-rec)
(trace mayor)

```

## Intérprete

```

Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
>(mayor '())
>(mayor ())
la lista vacía no forma parte del
conjunto de las listas de números
<#<void>
> (mayor '(1 2 3 4 5))
>(mayor (1 2 3 4 5))
>(mayor-rec 1 (2 3 4 5))
>(mayor-rec 2 (3 4 5))
>(mayor-rec 3 (4 5))
>(mayor-rec 4 (5))
>(mayor-rec 5 ())
<5
5
>

```

Cuarto ejercicio: Posición de un elemento de una lista.

1. Tarea del procedimiento: calcular el ordinal asociado con un elemento de una lista.
2. Separaremos la tarea en dos procedimientos. Uno será recursivo, mientras que el otro excluirá la posibilidad de la lista vacía, porque esta no tiene elementos.
3. Procedimiento recursivo:
  - a) tendrá tres argumentos: `elem`, `lista` y `contador`;
  - b) la recursión consistirá en recorrer la lista hasta que esté vacía. Esta será la condición de parada;

- c) en cada aplicación se comprobará si `elem` es igual que el primero de `lista`;
- d) si son iguales, se devuelve el valor de `contador`;
- e) si no, se hace la llamada recursiva con el resto de la lista y se incrementa `contador` en 1.

#### 4. Procedimiento no recursivo:

- a) hace la tarea de excluir la posibilidad de la lista vacía.
- b) aplica el procedimiento recursivo haciendo `contador = 0`, porque Scheme asocia el primer elemento de una lista con el ordinal 0.

Obsérvese que usamos `equal?`, para comparar `elem` y `(first lista)`, porque es el predicado más general de igualdad.

Editor

```
#lang scheme
(require mzlib/trace)
;; recursión

(define ordinal-elemento
  (lambda (elem lista)
    (if (null? lista)
        (printf "la lista vacía no
tiene elementos n")
        (ordinal-elem-rec elem lista
  0) )
  ))

(define ordinal-elem-rec
  (lambda (elem lista contador)
    (if (null? lista)
        (printf "~a no es un elemen-
to de ~a~n.elem lista)
        (if (equal? elem (first lista))
            contador
            (ordinal-elem-rec elem
(rest lista) (add1 contador)) )
        )
  ))

(trace ordinal-elemento)
(trace ordinal-elem-rec)
```

Intérprete

```
Bienvenido a DrScheme, versión
4.2.5 [3m].
Lenguaje: scheme.
> (ordinal-elemento "d" '( "a" "b"
1 2 3))
>(ordinal-elemento "d" ("a" "b" 1
2 3))
>(ordinal-elem-rec "d" ("a" "b" 1
2 3) 0)
>(ordinal-elem-rec "d" ("b" 1 2 3)
1)
>(ordinal-elem-rec "d" (1 2 3) 2)
>(ordinal-elem-rec "d" (2 3) 3)
>(ordinal-elem-rec "d" (3) 4)
>(ordinal-elem-rec "d" () 5)
d no es un elemento de ()
<#<void>
>(ordinal-elemento "d" '( "a" "d"
1 2 3))
>(ordinal-elemento "d" ("a" "d" 1
2 3))
>(ordinal-elem-rec "d" ("a" "d" 1
2 3) 0)
>(ordinal-elem-rec "d" ("d" 1 2 3)
1)
<1
1
>
```



# Unidad 10

## Un primer programa

Un programa es un conjunto de procedimientos y variables con cuya aplicación se realiza una tarea. La idea es la misma que hemos venido estudiando al definir procedimientos. Lo que cambia es la complejidad del texto existente en el editor.

### 10.1. Una calculadora

El objetivo es hacer un programa que simule una calculadora con las operaciones aritméticas básicas: sumar, restar, multiplicar y dividir. Los números han de poder tener decimales. Y existirá un botón de borrado. Algo como la figura 10.1.

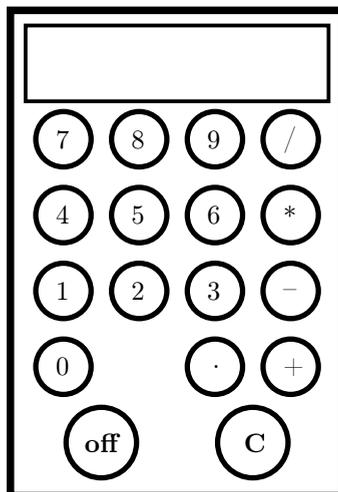


Figura 10.1: Calculadora

### 10.1.1. Sumadora

Empecemos haciendo un ejercicio más sencillo: una máquina de sumar.

1. Tarea del procedimiento: sumar los números que se introduzcan en el teclado.
2. Separaremos la tarea en dos procedimientos. Uno, cuya tarea consiste en obtener del teclado expresiones correctas. ¿Razón? Que en el teclado de la computadora hay signos que ni son cifras ni el resto de signos permitidos. Otro, que tomando la salida del anterior, la expresión correcta, realiza la operación solicitada.
3. Expresiones correctas:
  - a) para leer y recibir un *input* del teclado usaremos el procedimiento primitivo `read`;
  - b) usaremos una expresión `cond` para comprobar que lo leído por `read` es o no una expresión correcta;
  - c) si `input` es una variable local, la expresión `(input (read))` hace que lo leído por `read` sea el valor de `input`;
  - d) una vez hecho lo anterior, una expresión `cond` hace las comprobaciones necesarias y ejecuta las acciones correspondientes:

```
(cond
[1]  ((number? input) input)
[2]  ((eq? input '+) input)
[3]  ((eq? input '=) input)
[4]  ((eq? input 'c) input)
[5]  ((eq? input 'off) input)
[6]  (else ???)
)
```

Las líneas 1–5 son claras: el valor de `input` es una expresión *correcta*;

- e) la línea 6: a la cláusula `else` llega el proceso computacional cuando la expresión *no es correcta*. Por tanto, hay que obligar a que el proceso vuelva a comenzar. Por tanto, podemos subsumir las ideas anteriores en un procedimiento recursivo:

```
(define entrada
(lambda ()
(let ((input (read)))
(cond
((number? input) input)
((eq? input '+) input)
((eq? input '=) input)
((eq? input 'c) input)
((eq? input 'off) input)
))))
```

```

        (else (entrada))
      )
    )
  ))

```

Obsérvese que la variable `input` se declara dentro del texto del procedimiento mediante `let`. La cláusula `else`, caso de que la expresión no sea correcta, llama recursivamente al procedimiento `entrada`.

Por tanto, la *tarea* de este procedimiento es producir expresiones correctas, según lo especificado.

#### 4. Acciones según sea la expresión correcta:

- a) la idea aquí es escribir un procedimiento que sume los números que previamente se han escrito;
- b) usaremos una variable local `expression` cuyo valor será el devuelto por (`entrada`);
- c) y estableceremos las acciones mediante una expresión `cond`. Las comprobaciones serán:

```

(cond
 [1] ((number? expression) )
 [2] ((eq? expression '+) )
 [3] ((eq? expression '=) )
 [4] ((eq? expression 'c) )
 [5] ((eq? expression 'off) )
 )

```

Cuando la comprobación [1] se satisfaga, el número tecleado ha de ser añadido a la lista de números previamente introducidos. Por tanto, habremos de contar con una variable cuyo valor sea la lista de números, lista que inicialmente estará vacía.

Cuando la comprobación [2] sea satisfecha, lo que hay que hacer es volver a leer otra expresión desde el teclado (otro número).

Si la comprobación [3] se cumple, entonces hay que realizar la operación de sumar.

Si se cumple la [4], hay que borrar lo que hubiere en la memoria de la máquina.

El cumplimiento de la [5] apaga la máquina, es decir, sale del programa.

- d) Necesitamos, pues, una variable (de nombre `lista-numeros`) cuyo valor sea una lista de números.
- e) Podemos expresar estas ideas mediante el siguiente código:

```

(let ((expression (read))
      )
  (cond

```

```

((number? expression)
 (set! lista-numeros
      (cons expression lista-numeros))
 ... )
((eq? expression '+) ...)
((eq? expression '=)
 (apply + lista-numeros))
((eq? expression 'c)
 (set! lista-numeros empty)
 ... )
((eq? expression 'off) )
)
)

```

Obsérvese que la última cláusula carece de acción: porque cuando se cumpla, se acaba el programa.

- f) **apply** es un procedimiento primitivo que permite aplicar un procedimiento a una lista de argumentos. Su sintaxis es, en general:

```
(apply procedimiento lista-argumentos)
```

### 10.1.2. Una primera versión del programa

Esta versión del programa tendrá los siguientes elementos:

1. una variable global: **lista-numeros**.
2. un procedimiento para establecer expresiones correctas: **entrada**.
3. un procedimiento para hacer las distintas operaciones permitidas: introducir números, sumarlos, borrar y desconectar la sumadora.

La parte del programa que ya tenemos escrita es:

```

;;sumadora
(define lista-numeros empty)

(define entrada
  (lambda ()
    (let ((input (read)))
      (cond
        ((number? input) input)
        ((eq? input '+) input)
        ((eq? input '=) input)
        ((eq? input 'c) input)
        ((eq? input 'off) input)
        (else (entrada))
      )
    )
  )
)

```

```
)
))
```

Nos falta un procedimiento del que ya hemos visto lo esencial. Su código puede ser el siguiente:

```
(define opera
  (lambda ()
    (let ((expresion (entrada)))
      (cond
        ((number? expresion)
         (set! lista-numeros
              (cons expresion lista-numeros)))
        [1] (opera) )
        ((eq? expresion '+)
         [2] (opera))
        ((eq? expresion '=)
         [3] (apply + lista-numeros)
         [4] (set! lista-numeros empty)
         [5] (opera))
        ((eq? expresion 'c)
         [6] (set! lista-numeros empty)
         [7] (opera))
        ((eq? expresion 'off) )
        [8]
        [9] (else 'raro)
      )
    )
  ))
```

Analicemos las distintas cláusulas:

1. Las líneas 1, 2, 5 y 7 contienen una llamada recursiva a *opera*. ¿Por qué? Porque si la expresión es un número o  $+ o = o$  C, después de hacer las acciones correspondientes, hay que continuar con la tarea.
2. Cláusula `(number? expresion)`. Cuando la expresión es un número, hay que hacer dos cosas:
  - a) introducir el número en `lista-numeros`, es decir:
 

```
(set! lista-numeros (cons expresion lista-numeros)).
```
  - b) continuar la tarea: `(opera)`.
3. Cláusula `(eq? expresion '+)`. Cuando la expresión es el signo  $+$ , hay que continuar la tarea (normalmente escribiendo otro número):
  - a) `(opera)`.
4. Cláusula `(eq? expresion '=)`. Cuando la expresión es el signo  $=$ , hay que devolver la suma:

- a) (`apply + lista-numeros`).
  - b) mantener la sumadora lista para hacer otra suma, para lo cual hay que re-inicializar `lista-numeros`: (`set! lista-numeros empty`).
  - c) y continuar la tarea (`opera`).
5. Cláusula (`eq? expresion 'c`). En este caso hay que borrar lo que haya en memoria y continuar la tarea:
- a) (`set! lista-numeros empty`)
  - b) (`opera`)
6. Cláusula `else`. No parece necesaria, pero la mantenemos por si hubiéramos olvidado algo.

Si probamos ahora el texto del programa, nos encontraremos con la siguiente respuesta:

```
> (opera)
12
+
12
+
12
=
```

```
_____ eof
```

No obtenemos la respuesta, es decir: 36. ¿Por qué? Porque las acciones de la cláusula (`eq? expresion '=`) no están bien pensadas. En efecto, la secuencia de acciones hace:

1. (`apply + lista-numeros`) y
2. (`opera`)

pero *no escribe el resultado de sumar*. Para corregirlo, hay que escribir la primera acción así:

1. (`printf "~a~n" (apply + lista-numeros)`).

### 10.1.3. Segunda versión del programa con *letrec*

Como hemos visto, `let` permite definir variables locales dentro de un procedimiento. Recordemos la sintaxis general de `let`:

```
(let (
  (variable-local-1 valor-1)
  (variable-local-2 valor-2)
  ...
  (variable-local-n valor-n)
```

```

    )
  <let-body>
  )

```

Los valores de las variables pueden ser cualquier objeto o expresión correcta, según Scheme. Por tanto, una *expresión lambda* puede ser el valor de una variable local. Es decir, que las variables locales pueden tener como valor un procedimiento. Pero, si el procedimiento definido es recursivo, hay que usar una variante de `let`: `letrec`.

Esa idea puede ser usada para *encapsular* dentro de la definición de un procedimiento otro, aunque sea recursivo. Pero ¿por qué encapsular un procedimiento dentro de otro? La encapsulación —*sit venia verbo*— es una técnica que apareció en las bases de la programación orientada a objetos con el objetivo de evitar llamadas indeseables a un procedimiento, cuando el texto de un programa es largo y complejo. Una regla general de ese estilo de programar dice que un procedimiento subordinado a otro, en ejecución, sólo debe ser accesible para este otro. Este es nuestro caso: `opera` es un procedimiento que debería ser llamado por otro al que se subordina: `sumadora`.

Lo mismo, por otra parte, cabe decir de la variable global `lista-numeros`. Siguiendo la misma regla, incluiremos esta variable entre las variables de `letrec`.

El código de nuestro programa<sup>1</sup> podría quedar así:

```

(define entrada
  (lambda ()
    (let ((input (read)))
      (cond
        ((number? input) input)
        ((eq? input '+) input)
        ((eq? input '=) input)
        ((eq? input 'c) input)
        ((eq? input 'off) input)
        (else (entrada))
      )
    )
  ))

(define sumadora
  (lambda ()
    (letrec ((lista-numeros empty)
              (opera
               (lambda ()
                 (let ((expresion (read)))
                   (cond
                     ((number? expresion)

```

---

<sup>1</sup>¿No deberíamos, por lo mismo, incluir `entrada` en `sumadora`? Sí.



```

    )
  )
))
(define calculadora
  (lambda ()
    (letrec ((lista-numeros empty)
              (proc #f)
              (opera
                 (lambda (expr)
                   (cond
                     ((number? expr)
                      (set! lista-numeros
                             (cons expr lista-numeros))
                      (opera (entrada)) )
                     ((eq? expr '+)
                      (set! proc +)
                      (opera (entrada)))
                     [.] ((eq? expr '-)
                          (set! proc -)
                          (opera (entrada)))
                     [.] ((eq? expr '*)
                          (set! proc *)
                          (opera (entrada)))
                     [.] ((eq? expr '/')
                          (set! proc /)
                          (opera (entrada)))
                     ((eq? expr '=)
                      (printf "~a~n"
                               (apply proc lista-numeros))
                      (set! lista-numeros empty)
                      (opera (entrada)) )
                     ((eq? expr 'c)
                      (set! lista-numeros empty)
                      (opera (entrada)))
                     ((eq? expr 'off) )
                     (else 'mal)
                    )
                  )
                ))
              (opera (entrada))
            )
    ))
(calculadora)

```

Las cláusulas marcadas con [.] son las nuevas respecto al código de la sumadora. Otra novedad es el uso de otra variable local `proc`. Se usa para generalizar la idea de `apply`, puesto que el valor de `proc` dependerá de cuál sea la operación tecleada: (`apply proc lista-numeros`).

Parte II

**Inteligencia Artificial**



# Unidad 11

## Programa conexionista o subsimbólico

### 11.1. Introducción

Si el programa simbolista parte del presupuesto base consistente en defender que la mente humana lo que hace es trasegar símbolos, es decir, representaciones mentales, el programa conexionista tratará de explicar las funciones cognitivas intentando emular la configuración biológica del cerebro.

Los autores que siguen dicho planteamiento criticarán a los simbolistas sosteniendo que en el cerebro no encontramos representaciones mentales sino redes y capas de neuronas enlazadas entre sí. Por ello, partiendo del presupuesto monista que identifica mente y cerebro, será necesario explicar el objeto de estudio tomando como base los modelos biológicos, es decir, de dichas redes y capas de neuronas.

Si se comparase el funcionamiento entre computadores clásicos, entendiendo como tales “maquinas de Neumann-Turing”, y sistemas neuronales artificiales se podrían señalar algunas de las siguientes diferencias:

**Procesamiento en paralelo.-** Una red neuronal procesa la información de forma distribuida no en un proceso lineal y secuencial tal y como lo llevan a cabo los computadores clásicos.

**Tolerancia a los fallos.-** Lo anterior conlleva que, si una neurona de la red falla el sistema se adapta a dicho fallo al contrario de los computadores clásicos en el que cualquier tipo de error colapsa todo el sistema.

**Auto-organización.-** Las redes neuronales se autoorganizan pues en la fase de entrenamiento son capaces de modificar las intensidades de sus conexiones sinápticas a fin de dar la respuesta deseada.

**Aprendizaje adaptativo.-** Lo anterior implica también que tengan una gran capacidad de aprendizaje frente a los programas informáticos clásicos donde todo tiene que estar necesariamente previsto y programado.

Indudablemente las comparaciones entre ambos modelos se pueden alargar mucho señalando sus diferencias si bien las anteriores sirven para mostrar algunas de ellas.

## 11.2. Antecedentes históricos

Se puede considerar que el primer estudio sistemático sobre redes neuronales se debe a McCulloch y Pitts los cuales publicaron en 1943 y 1947 <sup>1</sup> el primero modelo computacional de una neurona biológica.

El trabajo de McCulloch y Pitts se centró básicamente en modelizar una neurona describiendo los elementos básicos de una única neurona artificial si bien nunca dieron el paso a construir redes de neuronas artificiales.

El siguiente paso relevante lo constituye el trabajo de Donald Hebb, el cual publicó en 1961 su obra titulada *La organización de la conducta* <sup>2</sup> en la cual aparece enunciado su famoso principio de aprendizaje por refuerzo. El conocido como *aprendizaje hebbiano* defiende que el peso o intensidad de una conexión sináptica se incrementa siempre que se activan simultáneamente el *input* y el *output* de una neurona.

Este modelo de aprendizaje propuesto por Hebb constituirá la base de todos los sistemas de aprendizaje por refuerzo de las distintas redes neuronales también denominados “aprendizaje no supervisado”.

El siguiente paso fue el dado por Rosenblatt el cual publicó un trabajo <sup>3</sup> centrado en lo que se denominará “aprendizaje supervisado”. Si en los modelos basados en el aprendizaje hebbiano no se tiene en cuenta la señal deseada, ahora, en estos nuevos modelos de aprendizaje será fundamental la salida que se quiere obtener. La diferencia entre la señal producida por la red neuronal y la señal deseada será el error de salida el cual será necesario ir corrigiendo disminuyendo dicho error hasta que ambos *outputs* coincidan en cuyo caso la red habrá “aprendido”. Dicha disminución del error se produce mediante sucesivas iteraciones en las que, en cada una de ellas, se lanza un patrón de *inputs* y en las sucesivas iteraciones se van corrigiendo el valor de los pesos sinápticos hasta que se encuentre una combinación de dichos pesos que produce el *output* buscado. A este tipo de redes con la regla de aprendizaje supervisado se les denominará “perceptrones” o “perceptrón de Rosenblatt”.

---

<sup>1</sup>W.W. McCulloch and W. Pitts, A Logical Calculus of the Ideas Imminent in Nervous Activity, *Bulletin of Mathematical Biophysics*, 5:115-133, 1943. W. Pitts and W.W. McCulloch, How We Know Universals, *Bulletin of Mathematical Biophysics*, 9:127-147, 1947.

<sup>2</sup>D.O. Hebb, *Organization of Behaviour*, Science Editions, New York, 1961.

<sup>3</sup>F. Rosenblatt, *Principles of Neurodynamics*, Science Editions, New York, 1962.

Por las mismas fechas, a primeros de los 60, Widrow y Hoff publicaron sus trabajos <sup>4</sup> proponiendo la que se denomina ADALINE (*ADaptive LINear Elements*) con el algoritmo de aprendizaje denominado LMS (*Least Mean Square*) dentro del grupo de sistemas de aprendizaje supervisado y basado en perceptrones.

Pero si el programa conexionista parecía avanzar con fuerza, el trabajo de Minsky y Papert <sup>5</sup> mostró qué clases de funciones no eran computables por los perceptrones lo cual supuso un serio revés para esta línea de investigación. Uno de los ejemplos de función no computable que pusieron de manifiesto Minsky y Papert fue la función XOR (disyunción exclusiva) dado que no es una función linealmente separable.

Si bien la crítica de Minsky y Papert supuso un parón importante en el desarrollo de este programa de investigación debido también a su enorme prestigio, la existencia de este problema ya era conocida por lo que se propuso el denominado “perceptrón multicapa” o MLP (*Multi Layer Perceptron*). No obstante, Minsky y Papert alegaban que incluso en el caso de perceptrones multicapa los algoritmos de aprendizaje no eran bien conocidos. Por todo ello la investigación dentro de esta línea se paralizó durante casi veinte años.

A pesar de dicha paralización, después del parón mencionado, se pudo demostrar <sup>6</sup> que empleando funciones sigmoideas como función de salida de cada neurona es posible llevar a cabo cualquier clase de clasificación. Es cierto, como argumentaban Minsky y Papert, que los perceptrones de una sola capa no funcionan más que como meros “discriminadores”. No obstante, con la introducción del algoritmo de aprendizaje conocido como “retropropagación” (*backpropagation*) y el empleo de funciones sigmoideas, el hecho es que las redes multicapa se convierten en “aproximadores” resolviendo cualquier problema de clasificación.

Se puede decir que el desarrollo del algoritmo de retropropagación superó el parón generado por la crítica de Minsky y Papert dando nuevo vigor al programa conexionista. Dicho algoritmo fue desarrollado de manera independiente por autores como Werbos, Rumelhart et al. (conocido como “Grupo PDP” -*Parallel Distributed Processing Research Group*-) y Parker <sup>7</sup> en fechas casi simultáneas.

---

<sup>4</sup>B. Widrow, M. E. Hoff, Adaptive Switching Circuits, In *IRE WESCON Convention Record*, pages 96-104, 1960.

<sup>5</sup>M. Minsky, S. Papert, *Perceptrons*, MIT press, Cambridge, MA, 1969.

<sup>6</sup>Teorema de Cybenko. G. Cybenko, Approximation by Superposition of a Sigmoidal Function, *Mathematics of Control, Signals, and Systems*, 2:303-314, 1989.

<sup>7</sup>P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*, PhD thesis, Harvard University, Boston, 1974. D.E. Rumelhart, G. E. Hinton and R. J. Williams, Learning Internal Representations by Error Propagation, In D. E. Rumelhart, J. L. McClelland and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations, pages 318-362, MIT Press, Cambridge, MA, 1986. D.B. Parker, Learning Logic, Technical report, Technical Report TR-47, Cambridge, MA: MIT Center for Research in Computational Economics and Management Science, 1985.

### 11.3. Descripción de un Sistema Neuronal Artificial

Todo Sistema Neuronal Artificial (SNA) está compuesto por los siguientes elementos:

**Neurona.-** Es el elemento básico. En este trabajo a cada una de ella se designará con la letra  $N$  y un subíndice por ejemplo  $N_1$ .

**Capa.-** La agrupación de varias neuronas unidas mediante conexiones entre ellas constituyen una capa de neuronas. Cada una de las posibles capas se designará la letra  $C$  y un subíndice por ejemplo  $C_1$ .

**Red.-** La agrupación de varias capas también unidas mediante conexiones entre las distintas capas forman una red. Cada una de dichas redes de designan mediante la letra  $R$  y un subíndice.

**Sistema.-** Si varias redes se conectan entre sí forman un sistema neuronal. Cada una de ellas se designará con la letra  $S$  y un subíndice.

A continuación se irán describiendo cada uno de estos elementos pormenorizadamente.

#### 11.3.1. Neuronas artificiales

En toda neurona artificial se puede considerar que se dan tres componentes básicos:

**Entradas.-** Cualquier neurona, como ocurre en las biológicas, recibe un conjunto de señales de entrada cada una de ellas con un valor concreto y con una intensidad determinada. De este modo se podría representar cada entrada mediante el par siguiente:  $\langle x_n, w_{i,j} \rangle$  donde cada elemento de dicha expresión tiene el siguiente significado:

$x_n$ .- El signo  $x$  indica el valor de la señal recibida de otra neurona anterior. El subíndice  $n$  identifica la neurona de la que procede la señal mencionada.

$w_{i,j}$ .- El signo  $w$  recoge la intensidad de la señal. Se emplea la letra  $w$  debido a que normalmente en la literatura conexionista se designa este valor con la palabra “peso”, en inglés *weight*. Los subíndices indican la neurona de origen y la de destino; por ejemplo, la expresión  $w_{h,i}$  indica que el “peso” de esta conexión es el valor recogido en  $w$ , y que dicho valor indica que dicha conexión enlaza la neurona  $N_h$  con la  $N_i$ .

Las señales, tanto entrantes como salientes, pueden ser a su vez de dos tipos teniendo en cuenta cuál sea su valor:

**Analógicas.-** Son aquellas que pueden tomar cualquier valor dentro de un rango determinado. Dichos rangos comunmente usados pueden ser los siguientes:  $[0, +1]$ ,  $[-1, +1]$ ,  $[-\infty, +\infty]$ .

**Digitales.-** Son aquellas cuyo valor únicamente puede tomar dos únicos valores, por ejemplo, 0 o 1, pero ninguno más.

**Computación.-** Cada neurona  $N_i$  recibe pues un conjunto de de señales las cuales son modificadas generando otra nueva señal de salida. Desde otro punto de vista se puede decir que cada neurona se encuentra en un estado inicial en el momento  $t$  cambiando a un nuevo estado en el momento  $t + 1$ .

**Salida.-** La neurona, una vez transformado el conjunto de señales entrantes, genera una única señal de salida con un valor determinado. A dicha señal de salida se la denomina “señal de salida”, “señal efectiva”, etc. designándola comúnmente con la letra  $y_1$ .

Como luego se verá con detalle, en realidad, la señal de salida de cualquier neurona artificial ( $y_i$ ) es la resultante de una composición de funciones, la de propagación, la de activación y la de salida. Expresado en forma matemática se podría escribir:

$$y_i = f_{\text{salida}}(f_{\text{activación}}(f_{\text{propagación}}))$$

A la vista de lo anterior se puede representar, en principio, cualquier neurona artificial tal y como aparece en la figura 11.1 de la página 97. A continuación se irán describiendo pormenorizadamente cada uno de dichos componentes.

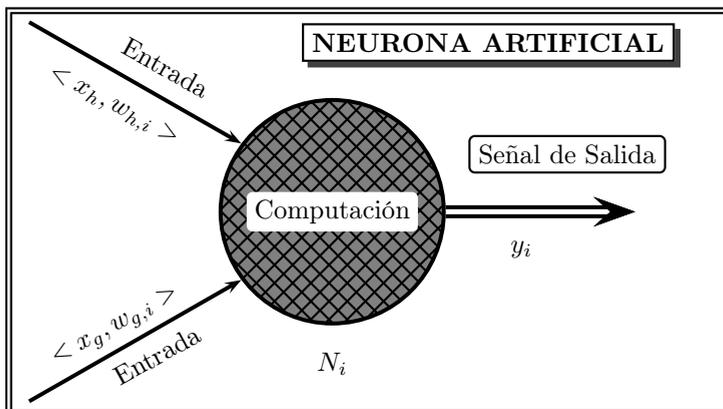


Figura 11.1: Ejemplo de una neurona artificial

### Función de propagación

Como ya se ha indicado, toda neurona recibe una o varias señales de entrada teniendo cada una de ellas, en general, un valor y una intensidad o “peso”. Por

eso, la primera cuestión consiste en calcular cuál es el valor total de las señales que recibe una neurona. Esto se calcula mediante la llamada “función de propagación” ( $f(net)$ ). Para calcular el valor de dicho conjunto de entradas se han elaborado diversas funciones siendo las principales las que se recogen en el cuadro 11.1 de la página 98.

Se suele designar a la función de propagación con el signo  $f(net)$  pues calcula el “neto” de todas las señales recibidas ponderadas de alguna manera por sus intensidades o “pesos”.

<b>FUNCIONES de PROPAGACIÓN</b>	
Sin umbral	Con umbral
(1) $f(net) = \sum_{i=1} x_i w_{h,i}$	$f(net) = \left( \sum_{i=1} x_i w_{h,i} \right) - \theta$
(2) $f(net) = \sqrt{\sum_{i=1} (x_i - w_{h,i})^2}$	$f(net) = \left( \sqrt{\sum_{i=1} (x_i - w_{h,i})^2} \right) - \theta$

Cuadro 11.1: Funciones de propagación

La expresión indicada con (1) se denomina “Función lineal” y la recogida con el (2) “Distancia euclídea”.

Es muy común añadir al cálculo de la función de activación un elemento más denominado “umbral” designado con la letra  $\theta$ . El motivo del por qué incluir un “umbral” se comprenderá mejor cuando se hable a continuación de la función de activación y de la de salida. En este momento se puede decir que el umbral sirve para indicar, en líneas generales, cuándo una neurona emite una señal de salida o no; si el valor del conjunto de señales de entrada supera el valor del umbral entonces se produce la señal de salida, en caso contrario la neurona no se activa.

En la literatura conexionista se suele emplear la palabra “bias” para designar al umbral.

Todo lo anterior se refiere a un momento temporal determinado que se designa con la letra  $t$  lo cual significa que, en el momento  $t$ , el valor de la función de activación de la neurona  $N_i$  es el calculado por la función  $f(net)$ .

### Función de activación

Antes de nada hay que indicar que en muchas obras sobre redes neuronales se mezclan los conceptos de “función de activación” y “función de salida”. Sin embargo aquí se considera que es más exacto y didáctico el diferenciar ambas. El motivo es que se pueden diseñar redes en las que el procesamiento que hacen las neuronas tenga en cuenta el estado que tenía la misma en el momento anterior  $t - 1$ . Pongamos un ejemplo.

Muchas veces, en la vida cotidiana, nos encontramos ante situaciones en la que vamos andando y alguien nos pisa un pie. Con educación el que nos ha pisado se disculpa, aceptamos sus disculpas y nos vamos con el pie dolorido. Un minuto después otra persona nos vuelve a pisar en el mismo pie. Vemos las estrellas y la persona se disculpa correctamente, aceptamos sus disculpas continuando la marcha. Al entrar en el metro, otra persona que entra precipitadamente nos vuelve a pisar y entonces no aguantamos más y le llamamos de todo. La gente que nos rodea se muestra sorprendida pues considera, con razón, que la respuesta es desproporcionada.

Pensando en todo el proceso se comprende fácilmente que la respuesta de un ente biológico no se encuentra determinada únicamente por el mecanismo de acción-reacción sino que interviene también el estado previo en el que se encuentre el sujeto en el momento anterior. La explicación de una conducta concreta, en muchas ocasiones, no depende del estado del individuo sino que interviene también el estado del mismo en el momento  $t - 1$  o momentos anteriores.

Por este motivo se puede diseñar un sistema en el que las neuronas, al computar las señales entrantes y generar la señal de salida, se tenga en cuenta el estado de dicha neurona en el momento anterior, es decir, en el momento  $t - 1$ .

Normalmente, la función de activación será la de igualdad, siendo su valor el mismo que el de la función de propagación pero se pueden considerar otras alternativas. Si a la función de activación la designamos con la expresión  $f(a_i)$  las dos posibilidades se recogen en la tabla 11.2 en la página 99.

<b>FUNCIONES de ACTIVACIÓN</b>	
Estado anterior	Estado actual
$f(a_i)(t) = f((a_i)(t - 1), f(net)(t))$	$f(a_i)(t) = f(net)(t)$

Cuadro 11.2: Funciones de activación

Como se puede comprobar simplemente hay que considerar si se quiere tener en cuenta el estado anterior de la neurona o no.

### Función de salida

Esta función es la que computa el valor de la señal de salida de la neurona  $N_i$ .

Las funciones de salida empleadas a la hora de configurar una neurona son las que se recogen en el cuadro 11.3 en la página 100.

A continuación se van a comentar las principales funciones de salida:

- 1) Función de Identidad.-** También se la denomina *purelin*<sup>10</sup>. Esta función no plantea ningún problema pues la función de salida es igual a la función de activación. Por tanto, sus valores son los mismos.

<sup>10</sup>Estas denominaciones alternativas son la forma en que se denominan estas funciones en el *Net-Toolbox* de *Matlab*.

## FUNCIONES de SALIDA

Nombre	Función	Rango
Identidad	$y = x$	$[-\infty, +\infty]$
De Escalón	1) $y = \text{signo}(x)$	$\{-1, +1\}$
	2) $y = H(x)$	$\{0, +1\}$
	3) $y = \begin{cases} -1 & \text{si } x < 0 \\ x & \text{si } 0 \leq x \leq 1 \\ +1 & \text{si } x > 1 \end{cases}$	$[-1, +1]$
Sigmoideas	1) $y = \frac{1}{l + e^{-x}}$	$[0, +1]$
	2) $y = \text{tgh}(x)$	$[-1, +1]$
Gaussiana	$y = A.e^{Bx^2}$	$[0, +1]^8$
Sinusoidal	$y = A.\text{sen}(\omega x + \phi)$	$[-1, +1]^9$

Cuadro 11.3: Funciones de salida

**2) Funciones de Escalón.-** Las funciones de escalón son muy habituales pudiendo ser de varios tipos:

**2.1) Limitador Fuerte.-** También se la conoce con el nombre de *hardlim*. El rango de esta función es  $\{0, +1\}$ . La expresión de la misma es la siguiente:  $y = \begin{cases} 0 & \text{si } x < 0 \\ +1 & \text{si } 0 \geq 0 \end{cases}$ . Para designar a esta función también se emplea la denominación de “función de Heaviside” en honor al físico y matemático Oliver Heaviside (1850–1925) empleando la nomenclatura  $y = H(\dots)$ . Esta es una de las funciones más empleadas dado que siempre genera señales digitales de salida (1 ó 0).

**2.2) Limitador Fuerte simétrico.-** También se la denomina *hardlims*. El rango de esta función es  $\{-1, +1\}$ . La expresión de la misma es la siguiente:  $y = \begin{cases} -1 & \text{si } x < 0 \\ +1 & \text{si } 0 \geq 0 \end{cases}$ .

**2.3) Lineal saturado.-** También se la denomina *satlin*. El rango de esta función es  $[0, +1]$ . La expresión de la misma es la siguiente:  $y = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } 0 \leq x \leq +1 \\ +1 & \text{si } x > 1 \end{cases}$ .

**2.4) Lineal saturado simétrico.-** También se la denomina *satlins*. El rango de esta función es  $[-1, +1]$ . La expresión de la misma es la siguiente:  $y = \begin{cases} -1 & \text{si } x < -1 \\ x & \text{si } -1 \leq x \leq +1 \\ +1 & \text{si } x > 1 \end{cases}$ .

**3) Funciones sigmoideas.-** Las funciones sigmoideas reciben este nombre debido a que su representación gráfica tiene la forma de una “s” tumbada. Se suelen emplear cuando lo que interesa obtener son salidas analógicas dentro de un rango de valores determinado. Precisamente las dos funciones sigmoideas se distinguen básicamente por el rango de valores que toman.

**3.1) Sigmoidea logarítmica.-** También se la denomina *logsig* o exponencial. El rango de esta función es  $[0, +1]$ . La expresión de la misma es la siguiente:  $y = \frac{1}{1 + e^{-x}}$ .

**3.2) Sigmoidea hiperbólica.-** También se la denomina *tansig*. El rango de esta función es  $[-1, +1]$ . La expresión de la misma es la siguiente:  $y = \text{tgh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ .

**4) Función Gaussiana.-** El rango de esta función es  $[0, +1]$ . La expresión de la misma es la siguiente:  $y = A.e^{Bx^2}$  o también  $f(x) = ae^{-\frac{x^2}{\sigma^2}}$ .

**5) Función Sinusoidal.-** El rango de esta función es  $[-1, +1]$ . La expresión de la misma es la siguiente:  $y = A.\text{sen}(\omega x + \phi)$ .

Si se observa la expresión matemática de las funciones de escalón anteriores se aprecia que siempre se baraja la posibilidad de que la señal de entrada sea mayor, menor que cero o se encuentre dentro del rango especificado. Este dato se suele modificar en muchas ocasiones considerando que el “disparo” o activación de la neurona depende de si el valor de la función de activación supera o no el valor el umbral de la neurona. Por este motivo, se pueden modificar dichas funciones según los intereses que se persigan.

Un ejemplo de modificar la función de salida del tipo escalón considerado el valor del umbral ( $\theta$ ) teniendo la siguiente forma:  $y = \begin{cases} 0 & \text{si } x < \theta \\ +1 & \text{si } 0 \geq \theta \end{cases}$ . La anterior función se denomina “función de escalón unitario” o de Heaviside.

El resumen de todo lo anterior se puede ver gráficamente en la figura 11.2 en la página 102.

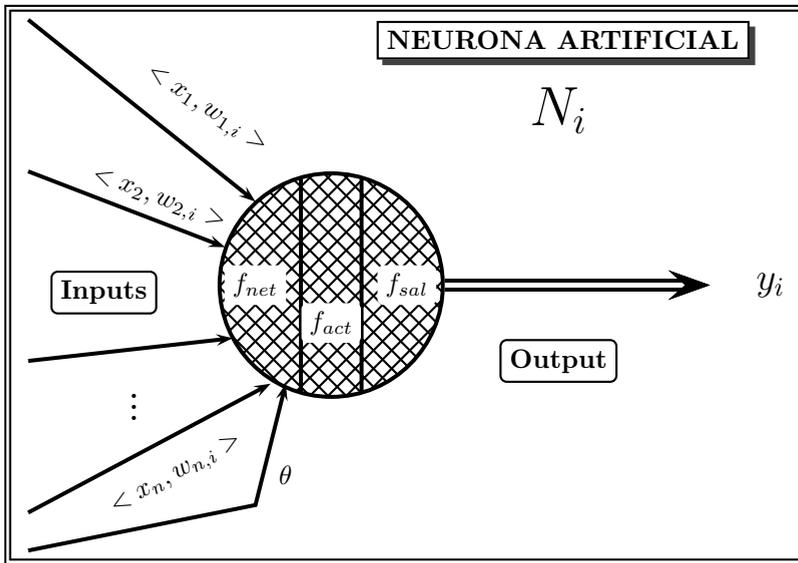


Figura 11.2: Ejemplo neurona artificial con sus funciones

### Clases de neuronas

Debido a las diversas combinaciones que pueden darse a la hora de configurar una neurona barajando el número de entradas, las diversas funciones de propagación, de salida, etc., se han venido empleando diversos nombres para identificarlas. Las principales clases de neuronas atendiendo a su configuración son las siguientes:

**Neurona McCulloch-Pitts.-** Es aquella neurona cuyos valores de salida únicamente pueden ser 0 o 1.

**Neuronas tipo Ising.-** Son aquellas neuronas que únicamente tienen como valores de salida +1 o -1 teniendo como función de salida la del signo de  $x$  ( $f(x) = \text{sign}(x)$ ). Lo que indica este tipo de neurona es si se encuentra activada (salida = +1) o desactivada (salida = -1).

**Neurona de tipo Potts.-** Son aquellas neuronas cuyos valores de salida son discretos y variados dentro de un rango determinado. Por ejemplo, -3, -2, -1, 0, +1, +2, +3.

Como se puede apreciar que el criterio de clasificación toma como base el tipo de salida que produce cada neurona tal y como se desee configurar.

### 11.3.2. Capas de neuronas

Una vez que se ha descrito la composición de una neurona, éstas se pueden agrupar entre sí por medio de conexiones formando diversas agrupaciones a las que se denominan “capas”.

Las distintas capas se suelen clasificar en tres grandes grupos:

**Capa de entrada.-** Es aquella capa de neuronas que reciben la información del exterior haciendo muchas veces la función de meros sensores que se limitan a pasar la información recibida a las capas posteriores. Un ejemplo de una capa de entrada podría ser la retina la cual transforma la luz recibida por el ojo transformándola en señales eléctricas hacia la zona occipital.

**Capa de salida.-** Es aquella capa de neuronas que transfieren las señales de la red hacia el exterior de la misma. Por ejemplo, en un sencillo robot la capa de salida emitiría una posible señal analógica que activaría la velocidad de un motor en cada una de sus cuatro ruedas.

**Capas ocultas.-** Son aquellas capas de neuronas que se encuentran situadas entre la capa de entrada y la de salida. Pueden ser ninguna, una o varias en función de la arquitectura de la red. Dependiendo de la función que se quiera que desempeñe la red cabe la posibilidad de que no sean necesarias dichas capas ocultas o, por el contrario, resulte imprescindible su existencia.

Un ejemplo de una red neuronal es el que aparece en la figura 11.3 de la página 104 donde se representa un perceptrón multicapa.

Para comprender el gráfico 11.3 se describen cada uno de sus elementos:

$x_i^\mu$ .- Designa el vector de los inputs o señales de entrada de la capa exterior.

$y_j^\mu$ .- Designa el vector de señales de entrada de la capa oculta los cuales proceden de la capa de entrada.

$z_k^\mu$ .- Designa el vector de señales de entrada de la capa de salida que proceden de la anterior capa oculta.

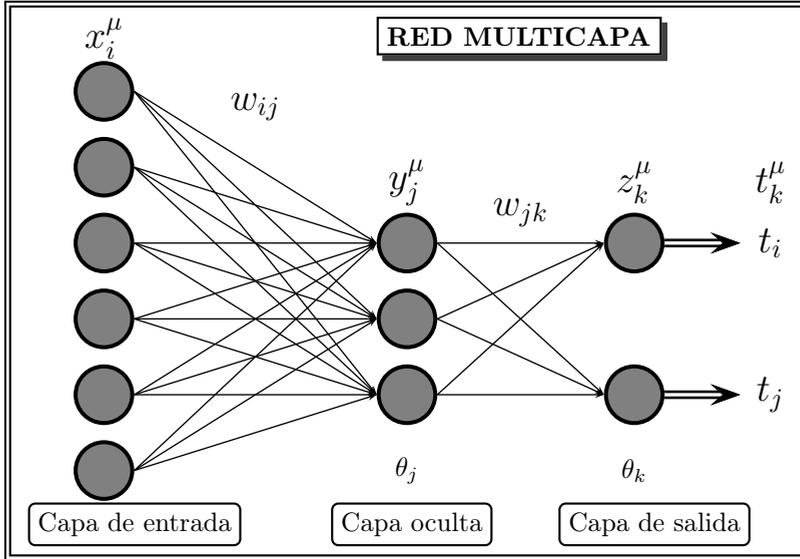


Figura 11.3: Perceptrón multicapa

$t_k^\mu$ .- Designa el vector de señales de salida de la capa de salida. Se emplea la letra “t” de la palabra inglesa *target* que significa “objetivo” pues es el patrón de señales que se desea produzca la red.

$w_{ij}$ .- Es la matriz de pesos de las conexiones entre la capa de entrada y la oculta.

$w_{jk}$ .- Designa la matriz de pesos de las conexiones entre la capa oculta y la de salida.

$\theta_j$  y  $\theta_k$ .- Designa el vector de valores del umbral en cada una de dichas capas.

Si como se ha visto una red neuronal se estructura en diferentes capas, todas las neuronas que forman cada capa se enlazan con las de la capa siguiente por medio de conexiones. Dichas conexiones no sólo pueden darse entre capas contiguas sino que cabe la posibilidad de que una misma neurona se conecte con otra tanto de la misma capa, de la siguiente como consigo mismo. Por este motivo la clase de conexiones determinan las distintas topologías de una red.

Desde ese punto de vista, atendiendo al tipo de conexiones que pueden tener las neuronas de cada capa, se pueden clasificar las mismas del siguiente modo:

**Conexiones crossbar.-** Son aquellas cuyas salidas se cruzan entre sí.

**Conexiones autorrecurrentes.-** Son aquellas en las que la salida de una neurona es a su vez entrada de ella misma.

La configuración de las conexiones entre las neuronas que forman las diferentes capas determinan el flujo de información que circula por la red neuronal. Intuitivamente uno se da cuenta de que la información entra a través de las neuronas de la capa de entrada, transcurre a través de las capas ocultas y sale al exterior mediante las neuronas de la capa de salida. La conclusión de esto es que la señal, en principio, es direccional, es decir, tiene un sentido de la marcha.

No obstante se han configurado redes en las que la señal que discurre por la red puede volver a entrar en la misma o llevar caminos más tortuosos. Por este motivo, teniendo en cuenta el sentido de la información que circula por la red se pueden clasificar la propagación de la señal del siguiente modo:

**Propagación hacia adelante (*feedforward*).**- Es aquella en la que ninguna salida de las neuronas que forman una capa es entrada de cualquier otra neurona de la misma capa. Redes configuradas con este tipo de conexiones son útiles para el reconocimiento o clasificación de patrones. Ejemplos de ellas serían el “Perceptrón”, “Adaline”, “Madaline”, entre otras.

**Propagación hacia atrás (*feedback*).**- Es aquella en la que alguna salida de las neuronas que forman una capa es a su vez entrada de alguna neurona de la misma capa o de alguna perteneciente a una capa de un nivel anterior. De este modo la información circula tanto hacia adelante (*forward*) como hacia atrás (*feedback*).

En este tipo de conexiones existen dos grupos de pesos: a) los de las conexiones *forward* y los de las conexiones *feedback*. Ejemplos de redes que emplean este tipo de conexiones serían la ART (*Adaptative Resonance Theory*) y la BAM.

Un ejemplo de una red neuronal con capas ocultas y conexiones con propagación “hacia adelante” es la que se representa en la figura 11.3 de la página 104.

## 11.4. Aprendizaje en el programa conexionista

Quizás una de las cuestiones más interesantes del programa conexionista radique en la explicación de la pregunta acerca de qué significa “aprender” así como intentar explicar cómo se produce dicho proceso en el ser humano.

Desde el punto de vista de la Psicología se han distinguido varias clases de aprendizajes como el aprendizaje por condicionamiento en sus diversas variantes, el aprendizaje vicario o por imitación, etc. Pero la cuestión sigue en pie, ¿cómo se producen en el ser humano dichos aprendizajes?

### 11.4.1. Concepto de aprendizaje

La idea fundamental que subyace a todas las clases de aprendizaje o entrenamiento de redes neuronales consiste en una idea básica y sencilla, a saber.

Aprendemos modificando los pesos de las conexiones

Como ya se ha podido ver por todo lo indicado previamente, cualquier red recibe un patrón de señales como *inputs* a través de su capa de entrada. Dicho patrón se traslada a la siguiente capa teniendo en cuenta que los *inputs* de la siguiente capa tienen en cuenta el valor de los pesos sinápticos o, en otras palabras, la intensidad de dichas conexiones. La capa correspondiente procesa el patrón de señales trasladándola sucesivamente a la capa siguiente hasta llegar a la de salida lo cual genera el patrón de *outputs* de la red.

Obviamente dichas señales de salida no son la que deseáramos dado que no producen el efecto deseado. Por ejemplo, si se quiere que un red sencilla que evalúe la tabla semántica de la conjunción lógica (AND) tendremos que cada vez que el patrón de entrada sea (1,1), (1,0), (0,1), (0,0) nos dé el valor deseado, es decir, (1) en el primer caso y (0) en todos los demás. Si esto no se produce será preciso “entrenar” a la red o conseguir que ésta “aprenda”.

El problema se resuelve cuando encontremos una combinación de pesos sinápticos que haga que la red produzca el *output* deseado. En ese caso diremos que la red ha “aprendido” o está “entrenada”.

De modo análogo, el ser humano aprendería modificando las intensidades de las conexiones entre nuestras neuronas biológicas y así quedaría explicado en qué consiste aprender algo.

Pero el ejemplo anterior se puede considerar que es simplemente parcial. Se ha partido de la base de que se conoce la respuesta deseada lo cual es cierto en muchas ocasiones pero también es verdad que nos encontramos ante situaciones en las que no conocemos dichas respuestas y hemos aprendido, por ejemplo, a clasificar las cosas. Una muestra de esto último es lo que se denomina “categorizar”. Constantemente vemos cosas tales como “este coche”, “aquel otro coche”, “aquel otro”, etc. y, a pesar de la ingente diversidad de modelos y marcas, si alguien nos pregunta qué esa cosa siempre contestamos que son “coches”. En otras palabras, subsumimos cosas diversas bajo un único concepto. ¿Cómo lo hace nuestro cerebro?

Con el fin de abarcar las distintas clases de aprendizaje que se han desarrollado en el ámbito de las redes neuronales artificiales se puede hacer la siguiente clasificación:

**Aprendizaje supervisado.-** Esta clase de aprendizaje es aquel en el que se conoce la respuesta deseada.

**Aprendizaje no supervisado.-** En esta clase de aprendizaje se desconoce la respuesta deseada siendo la propia red la que agrupa las señales entrantes en diversas categorías.

**Aprendizaje por refuerzo.-** En esta clase de aprendizajes el proceso que se sigue es el de ensayo-error.

#### 11.4.2. Aprendizaje supervisado

Esta clase de aprendizaje es aquel en el que se conoce la respuesta deseada. Debido a que la respuesta efectiva producida por la red es diferente de la respuesta

deseada, la diferencia entre ambas es lo que se denomina “error”. Los algoritmos de aprendizaje de esta clase buscarán minimizar sucesivamente dicho error hasta que el mismo sea tan pequeño que se considere aceptable. Cuando se alcance dicho momento se considera que la red ha sido entrenada o ha aprendido. En esta clase de aprendizajes se habla de “supervisor” pues es el que supervisa si la respuesta efectiva coincide o no con la deseada.

Los algoritmos empleados en este tipo de aprendizaje son los siguientes:

**Regla Delta.-** Este algoritmo de aprendizaje fue desarrollado por Widrow y Hoff<sup>11</sup>. Esta regla es conocida también con los nombres de “Regla de Widrow-Hoff” o “Regla LMS” (*Least Mean Squares* -mínimos cuadrados-). Es el empleado en la red desarrollada por dichos autores conocida con el nombre de ADALINE (ADaptive Linear Neuron) la cual emplea una única neurona con función de salida lineal. El motivo del nombre de este algoritmo es que en cada iteración va minimizando la suma de los cuadrados de los errores, es decir, la diferencia entre la señal deseada y la efectiva:  $E(w) = \frac{1}{2} \sum_{j=1}^N (d_j - z_j)^2$ .

**Regla Delta generalizada.-** Este algoritmo, también conocido como algoritmo de retropropagación (*backpropagation*), fue desarrollado por Rumelhart y McClelland<sup>12</sup>.

Este algoritmo es empleado en el denominado “Perceptrón multi-capas” (MLP -*Multi-Layer Perceptron*-) que es precisamente aquel que incorpora capas ocultas.

**Algoritmos LVQ.-** Este algoritmo de aprendizaje fue desarrollado por Kohonen<sup>13</sup>.

### 11.4.3. Aprendizaje no supervisado

En esta clase de aprendizaje no se conoce la respuesta deseada por lo que la red lleva a cabo una correcta clasificación de los inputs en diferentes grupos. Se puede decir que la red “categoriza”. Durante la fase de entrenamiento los pesos se van modificando hasta que los patrones de entrada se agupan siguiendo patrones similares.

El algoritmo de este tipo de aprendizaje principal es el siguiente:

**ART.-** Este algoritmo, acrónimo de *Adaptive Resonance Theory*, fue desarrollado por Kohonen<sup>14</sup>

<sup>11</sup>Widrow B y Hoff M E, (1960), “Adaptive switching circuits”, *Proc. 1960 IRE WESCON Convention Record*, Part 4, IRE, New York, 96-104.

<sup>12</sup>Rumelhart D y McClelland J, (1986), *Parallel distributed processing: exploitations in the microstructure of cognition*, volumes 1 and 2, Cambridge: MIT Press.

<sup>13</sup>Kohonen T, (1989), *Self-Organising and Associative Memory* (3rd ed.), Berlin: Springer-Verlag.

<sup>14</sup>Mismo que el anterior.

Este tipo de aprendizaje se implementa en los denominados “Mapas auto-organizativos” (*Self-Organized Feature Map*) conocidos por la abreviatura SOFM desarrollados por Kohonen así como por Hopfield. El proceso de aprendizaje conlleva que determinadas neuronas sean “vencedoras” agrupando patrones de entradas en función de sus similitudes llevando a cabo categorizaciones.

#### 11.4.4. Aprendizaje por refuerzo

En este tipo de aprendizaje la red va alcanzando el objetivo deseado mediante un proceso de acierto y error de tal modo que la decisión tomada le acerca al objetivo los pesos sinápticos se incrementan reforzando dicha tendencia mientras que si le aleja de su meta dichos pesos disminuyen penalizando dicha tendencia.

Uno de los sistemas más conocidos de este sistema es el denominado *Q-learning* desarrollado por Sutton y Barton <sup>15</sup>. En dicho sistema la red posee una tabla, la denominada “Tabla-Q” que recoge la totalidad de posibles estados. Cada uno de dichos estados tiene dos posibilidades que almacenan los pesos sinápticos posibles los cuales van actualizándose cada vez que la red “toma una decisión” en función del resultado de la elección tomada. De este modo la red va “explorando” hasta alcanzar el objetivo deseado si bien se puede decir que es autónoma.

### 11.5. Construyendo una red

Pensemos una neurona ( $N_i$ ) con tres señales de entrada, umbral y una salida cuya representación gráfica aparece en la figura 11.4 de la página 108.

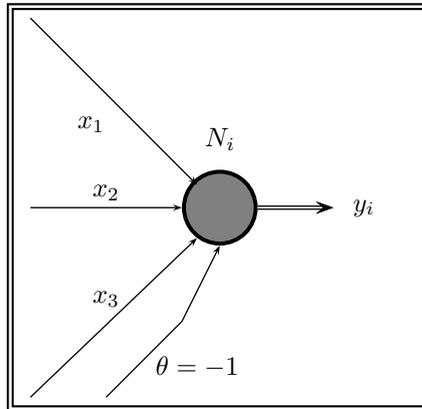


Figura 11.4: Construyendo una red

El primer paso es calcular la función de propagación ( $f(net)$ ). En este sentido tenemos dos vectores, el de señales de entradas y el de pesos de cada una de dichas entradas:

<sup>15</sup>Sutton R.S. y Barto A.G. (1998). *Reinforcement Learning. An Introduction*. MIT.

1.  $\overrightarrow{entradas} = [x_1, x_2, x_3]$ .
2.  $\overrightarrow{pesos} = [w_{1,i}, w_{2,i}, w_{3,i}]$ .

Como función de propagación se va a emplear:

$$f(net) = \sum_{i=1}^n x_i * w_{h,i} + \theta$$

Esta función de propagación es equivalente a <sup>16</sup>:

$$f(net) = \overrightarrow{entradas} * \overrightarrow{pesos} + \theta$$

No obstante, si fuese necesario emplear una multiplicación matricial tendríamos la siguiente equivalencia:

$$f(net) = \overrightarrow{entradas} * \overrightarrow{pesos} = \overrightarrow{entradas} * \overrightarrow{pesos}^T + \theta$$

Ahora veamos un ejemplo concreto de lo anterior.

1.  $\overrightarrow{entradas} = [1, 3, -5]$ .
2.  $\overrightarrow{pesos} = [4, -2, -1]$ .

Sea el umbral:  $\theta = -1$

Entonces:

$$\begin{aligned} f(net) &= \overrightarrow{entradas} * \overrightarrow{pesos} + \theta \\ &= [1, 3, -5] * [4, -2, -1] + -1 \\ &= (1 * 4 + 3 * -2 + -5 * -1) + -1 \\ &= 3 + -1 \\ &= 2 \end{aligned}$$

Expresemos ahora lo mismo pero empleando una multiplicación matricial:

$$\begin{aligned} f(net) &= \overrightarrow{entradas} * \overrightarrow{pesos} + \theta \\ &= \overrightarrow{entradas} * \overrightarrow{pesos}^T + \theta \\ &= [1, 3, -5] * \begin{bmatrix} 4 \\ -2 \\ -1 \end{bmatrix} + -1 \\ &= (1 * 4 + 3 * -2 + -5 * -1) + -1 \\ &= 3 + -1 \\ &= 2 \end{aligned}$$

Esta explicación distinguiendo entre una multiplicación entre vectores y una multiplicación matricial tiene importancia a la hora de transcribir todo lo anterior

---

<sup>16</sup>Aquí se contempla la multiplicación de dos vectores (en inglés *dot product*).

a cualquier lenguaje de programación evitando que se produzcan errores en tiempo de ejecución muy difíciles de detectar <sup>17</sup>.

Una vez que se ha calculado el valor de la función de propagación se puede considerar en este ejemplo tan simple que la función de activación es igual que la de propagación calculando pues únicamente la función de salida.

En este caso la función de salida sería: 
$$y = \begin{cases} 0 & \text{si } x < \theta \\ +1 & \text{si } x \geq \theta \end{cases}$$

Como en el ejemplo presente  $f(net) = 2$ , entonces la señal efectiva de la neurona  $N_i$  será  $y_i = +1$ .

---

<sup>17</sup>Si se quisiera emplear *Matlab* para constatar la corrección de cualquier cálculo que se haga, el producto de vectores se escribe  $dot(a,b)$  siendo  $a$  y  $b$  dos vectores de la misma longitud. Si se desea transponer un vector en forma de vector columna se emplea la expresión  $transpose(b)$  siendo  $b$  un vector fila.