

Alejandro Guerra-Hernández

Metodologías de Programación II

Programación Funcional

21 de febrero de 2011



Universidad Veracruzana

Departamento de Inteligencia Artificial
Sebastián Camacho No. 5, Xalapa, Ver.,
México 91000

Prefacio

Este texto está constituido por las notas del curso de Metodología de Programación II, en la Maestría en Inteligencia Artificial de la Universidad Veracruzana. El tema central de este curso es la **programación funcional**, y complementa al curso de Metodología de Programación I, cuyo tema es la programación lógica. El material está organizado para ofrecer un curso de 45 horas, durante un cuatrimestre, con dos sesiones de dos horas por semana.

Objetivos

El **objetivo** de este texto es introducir al estudiante a las técnicas y métodos de la programación funcional en el contexto de la Inteligencia Artificial. Para ello, la primera parte del mismo se ha organizado como un taller práctico de programación en Lisp; mientras que la segunda parte hace énfasis en el aspecto teórico de este paradigma de programación, ilustrado con ejemplos en el dialecto de ML conocido como OCaml.

Evaluación

El estudiante será **evaluado** conforme a su participación en clase, la calidad de sus tareas y el resultado que obtenga en el examen final o en la elaboración de un proyecto. La nota final, será calculada, aproximadamente, de la siguiente forma:

- 4 tareas para el 60 % de la nota final.
- La participación en un debate y/o exposición, así como la participación durante el curso para el 10 % de la nota final.
- El examen final o proyecto para el 30 % de la nota final.

Para obtener una **nota aprobatoria**, el alumno deberá haber obtenido notas aprobatorias en cada uno de las tareas a realizar, el debate y el examen final, i.e., no aprobar cualquier elemento de la evaluación, implica no aprobar el curso.

Tareas

Durante el curso, se asignarán cuatro **tareas**. Las tareas deberán entregarse a las 11:00 am del día designado (generalmente al encargarse la tarea siguiente, por ejemplo, la tarea T1 se entregará el día que será encargada la tarea T2, y así sucesivamente). El mérito del trabajo decrece 25 % por cada 24 horas de retraso. Las tareas pueden requerir investigación bibliográfica y experimentación en la computadora (caml, Ocaml, Lisp). Todos los trabajos deberán incluir al pie de cada página: i) el nombre del estudiante; ii) el nombre del curso; iii) el número de la tarea; y iv) el número de la página. Sólo se acepta papel carta 8'x11'. En todas las partes que involucran código, éste deberá ser documentado apropiadamente. Los trabajos son evaluados de la siguiente manera: Para que un ejercicio o pregunta reciba créditos, más del 50 % deberá estar resuelta de manera correcta o completa (es más reduitable invertir el tiempo en contestar una pregunta de manera correcta y completa, que responder a dos de manera parcial).

La discusión entre los estudiantes es uno de los aspectos más enriquecedores en un programa de maestría. Es sumamente importante que discutan entre ustedes las tareas, pero toda discusión al respecto deberá ser de manera oral. En ningún momento podrá un estudiante consultar las notas de trabajo escrito de otro. Cualquier anomalía en este sentido, causa **expulsión definitiva** del curso. La fecha de entrega de las tareas aparece en la sección de calendario de este documento.

Debates y Exposiciones

Cada alumno deberá participar en una **exposición y/o debate**, durante el curso. Tanto las exposiciones, como los debates, están relacionados con lecturas suplementarias al contenido del curso. En el caso de los debates, la postura a defender será asignada por el instructor del curso, quién fungirá como moderador. El ganador será designado por el público asistente (en principio, sólo los asistentes al curso).

Examen final

El **examen final** tendrá una duración máxima de dos horas y será de carácter escrito e individual. La fecha del mismo será a finales de febrero o principios de marzo, dependiendo del calendario oficial de nuestra universidad.

Material del curso

El estudiante es libre de elegir la implementación de Lisp que crea conveniente, aunque se recomienda usar **LispWorks 6.0**¹ para el proyecto final del curso (contamos con una licencia comercial en OS X). Otras implementaciones disponibles incluyen: **Closure**², **Allegro**³ y **SBCL**⁴, entre otras. Con respecto a ML utilizaremos el dialecto **OCaml 3.12.0**⁵.

Se les solicita usar **Latex**⁶ para generar los reportes con los resultados de sus tareas. Tanto Latex, como Lisp y ML pueden integrarse al editor **Emacs**⁷. El archivo de configuración `.emacs` debería contener algunas líneas como las siguientes:

```

1  ;; slime (setting Lisp for aquaemacs)
2
3  (add-hook 'slime-mode-hook 'imenu-add-menubar-index)
4
5  (setq inferior-lisp-program "~/Documents/lisp/lispworks"
6        lisp-indent-function 'common-lisp-indent-function
7        common-lisp-hyperspec-root
8        "file:///Applications/LispWorks%205.1/Library/lib/5-1-0-0/manual/online
9        /web/CLHS/"
10       slime-start-up-animation t)
11
12 (setq auto-mode-alist
13       (cons '("\\.lsp" . lisp-mode) auto-mode-alist))
14
15 ;; configuration for Ocaml
16
17 (add-to-list 'load-path "/Users/aguerra/Documents/elisp/tuareg-mode-1.45.6/")
18
19 (setq auto-mode-alist (cons '("\\.ml\\w?" . tuareg-mode) auto-mode-alist))
20 (autoload 'tuareg-mode "tuareg" "Major mode for editing Caml code" t)
21 (autoload 'camldebug "camldebug" "Run the Caml debugger" t)
22 (if (and (boundp 'window-system) window-system)
23     (require 'font-lock))

```

La línea 3 añade una entrada al menú de Emacs que lista las funciones definidas por el usuario. La línea 5 especifica la implementación de Lisp que se va a usar. La línea 8 define el camino a la documentación de Lisp en formato HTML. Slime y Tuareg-Mode deben cargarse de internet por separado.

¹ <http://www.lispworks.com>

² <http://trac.closure.com/ccl>

³ <http://www.franz.com/>

⁴ <http://www.sbcl.org/>

⁵ <http://caml.inria.fr/>

⁶ <http://www.latex-project.com>

⁷ <http://www.gnu.org>

Calendario

Este año las sesiones se llevarán a cabo los martes y los jueves de 11:00 a 13:00 hrs. El contenido y **calendario** de las sesiones es el siguiente:

Fecha	Tema	Tarea
06/12/2010	Introducción a la Programación Funcional	
08/12/2010	Introducción a Lisp 1	
13/12/2010	Introducción a Lisp 2	
10/01/2011	Recursividad y Listas en Lisp	T1
12/01/2011	Macros 1	
17/01/2011	Macros 2	
19/01/2011	Aspectos no funcionales de Lisp	T2
24/01/2011	Práctica 1: Un filtro de Spam	
26/01/2011	Práctica 2: Programación Web	
31/01/2011	Práctica 3: Árboles de decisión	T3
02/02/2011	Introducción a Ocaml	
07/02/2011	Listas y tipos de datos definidos por el usuario	
09/02/2011	Computaciones repetitivas	
14/02/2011	Tipos de datos concretos y recursivos	T4
16/02/2011	Cálculo lambda 1	
21/02/2011	Cálculo lambda 2	
16/03/2011	Entrega de Proyectos	

Referencias básicas

1. E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications With Objective Caml*. O'Reilly, Paris, France, 2000.
2. P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall International, 1993.
3. P. Graham. *ANSI Common Lisp*. Prentice Hall Series in Artificial Intelligence. Prentice Hall International, 1996.
4. W. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer-Verlag, Berlin Heidelberg New York, 2005.
5. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kauffman Publishers, 1992.
6. C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, UK, 1996.
7. P. Seibel. *Practical Common Lisp*. Apress, USA, 2005.

Xalapa, Ver., México
Diciembre 2010

Alejandro
Guerra-Hernández

Índice general

1. Introducción	1
1.1. ¿Porqué la programación funcional es relevante?	1
1.2. Funciones puras	2
1.3. Transparencia referencial	8
1.4. Funciones de orden superior	9
1.5. Recursividad	12
1.6. Consideraciones	13
Parte I Lisp	
2. Introducción a Lisp	17
2.1. Expresiones	17
2.2. Evaluación	18
2.3. Datos	19
2.4. Operaciones con listas	20
2.5. Valores de verdad	21
2.6. Funciones	22
2.7. Recursividad	23
2.8. Leyendo y escribiendo Lisp	24
2.9. Entradas y salidas	25
2.10. Variables	26
2.11. Asignaciones	27
2.12. Programación funcional	28
2.13. Iteración	29
2.14. Funciones como objetos	31
2.15. Tipos	33
2.16. Consideraciones	33
3. Listas en Lisp	35
3.1. Conses	35
3.2. Cons e igualdad	37

3.3.	Construyendo listas	38
3.4.	Ejemplo: compresión de datos <i>run-length</i>	39
3.5.	Funciones básicas	40
3.6.	Mapeos	42
4.	Macros en Lisp	45
4.1.	¿Cómo funcionan las macros?	45
4.2.	Backquote	47
4.3.	Definiendo macros simples	49
4.4.	Probando la expansión de las macros	50
4.5.	Ejemplos	52
5.	Una aplicación sencilla: Significancia de los codones en el DNA	53
5.1.	Paquetes	53
5.2.	La resolución del problema	54
5.3.	Una interfaz gráfica	58
5.4.	Creando un ejecutable	62
6.	Una aplicación más elaborada: ID3 en Lisp	63
6.1.	Árboles de decisión	63
6.2.	Representación de los ejemplos de entrenamiento	65
6.3.	Clasificación a partir de un árbol de decisión	67
6.4.	El algoritmo básico de aprendizaje de árboles de decisión	68
6.5.	Cargando ejemplos de entrenamiento	69
6.5.1.	Formatos de los archivos	69
6.5.2.	Ambiente de aprendizaje	70
6.5.3.	Lectura de archivos	70
6.6.	Librerías ASDF instalables	73
6.6.1.	Instalación de ASDF	74
6.6.2.	Instalación de ASDF-install	74
6.6.3.	Uso de librerías ASDF instalables	76
6.6.4.	Definiendo una librería ASDF: cl-id3	77
6.7.	Paquetes: cl-id3	77
6.8.	¿Qué atributo es el mejor clasificador?	78
6.8.1.	Particiones	78
6.8.2.	Entropía y ganancia de información	79
6.9.	Interfaz gráfica para cl-id3	85
6.9.1.	Definiendo la interfaz	86
6.9.2.	Definiendo el comportamiento de la interfaz	88
7.	Multi-procesamiento en Lisp	93
7.1.	Introducción	93
7.2.	Conceptos básicos	95
7.3.	Un ojo a los procesos	96
7.4.	Cerraduras y multi-procesos	98
7.5.	Esperas	99

Índice general	XI
7.6. Buzones	100
7.7. Hilos e interfaz gráfica	101
7.8. Locks	103

Parte II Teoría

8. Otro mundo es posible: AL	107
8.1. Introducción	107
8.2. Sintaxis de las expresiones en AL	108
8.3. Evaluación de las expresiones en AL	110
9. El Cálculo-λ	117
9.1. Introducción	117
9.2. Notación del Cálculo- λ	118
9.3. β -reducción y α -conversión	119
9.4. Un esquema de indexación para variables ligadas	124
9.5. Secuencias de reducción	129
Referencias	133

Capítulo 1

Introducción

Resumen El tema de este primer curso de metodologías de programación es la programación funcional. En este primer capítulo se presenta un panorama general de este paradigma de programación, con el objetivo de que ustedes puedan responder ¿Porqué es relevante estudiar la programación funcional? Para ello revisaremos conceptos como función, transparencia referencial, funciones de orden superior y recurrencia. Cuando se considere necesario, ilustraremos estos conceptos con código en Ocaml o en Lisp, los lenguajes que utilizaremos en este curso.

1.1. ¿Porqué la programación funcional es relevante?

Resulta curioso que cuando explicamos las ventajas de la programación funcional sobre otros paradigmas de programación, lo hacemos en términos negativos, resaltando cuales son las prácticas de otros paradigmas de programación que **no** están presentes en la programación funcional:

- En programación funcional no hay instrucciones de asignación;
- La evaluación de un programa funcional no tiene efectos colaterales; y
- Las funciones pueden evaluarse en cualquier orden, por lo que en programación funcional no hay que preocuparse por el flujo de control.

Siguiendo la propuesta de Hughes [10], este capítulo se centra en presentar las ventajas de la programación funcional en términos positivos. La idea central es que las **funciones de orden superior** y la **evaluación postergada**, son herramientas conceptuales de la programación funcional que nos permiten descomponer problemas más allá del **diseño modular** que inducen otros paradigmas de programación, como la estructurada. La evasión de la asignación, los efectos colaterales y el flujo de control son simples medios para este fin.

El capítulo se organiza de la siguiente manera. Primero presentaremos una serie de conceptos muy básicos sobre **funciones puras** y la composición de las mismas

como parte del diseño modular de programas funcionales. A continuación profundizaremos en esta idea a través de los conceptos de **interfaz manifiesta**, **transparencia referencial**, **función de orden superior**, y **recurrencia**.

1.2. Funciones puras

En matemáticas una función provee un mapeo entre objetos tomados de un conjunto de valores llamado **dominio** y objetos en otro conjunto llamado **codominio** o **rango**.

Ejemplo 1 *Un ejemplo simple de función es aquella que mapea el conjunto de los enteros a uno de los valores en $\{pos, neg, cero\}$, dependiendo si el entero es positivo, negativo o cero. Llamaremos a esta función *signo*. El dominio de *signo* es entonces el conjunto de los números enteros y su rango es el conjunto $\{pos, neg, cero\}$.*

Podemos caracterizar nuestra función mostrando explícitamente los elementos en el dominio y el rango y el mapeo que establece la función. A este tipo de caracterizaciones se les llama por **extensión**.

Ejemplo 2 *Caracterización de la función *signo* por extensión:*

$$\begin{aligned} & \vdots \\ \text{signo}(-3) &= \text{neg} \\ \text{signo}(-2) &= \text{neg} \\ \text{signo}(-1) &= \text{neg} \\ \text{signo}(0) &= \text{cero} \\ \text{signo}(1) &= \text{pos} \\ \text{signo}(2) &= \text{pos} \\ \text{signo}(3) &= \text{pos} \\ & \vdots \end{aligned}$$

También podemos caracterizar una función a través de reglas que describan el mapeo que establece la función. A esta caracterización se le llama por **intensión** o **intensional**.

Ejemplo 3 *Caracterización intensional de la función *signo*:*

$$\text{signo}(x) = \begin{cases} \text{neg} & \text{si } x < 0 \\ \text{cero} & \text{si } x = 0 \\ \text{pos} & \text{si } x > 0 \end{cases}$$

En la definición intensional de *signo*/1, x se conoce como el **parámetro formal** de la función y representa cualquier elemento dado del dominio. El cuerpo de la

regla simplemente específica a que elemento del rango de la función, mapea cada elemento del dominio. La regla que define *signo/1* representa un conjunto infinito de ecuaciones individuales, una para cada valor en el dominio. Debido a que esta función aplica a todos los elementos del dominio, se dice que se trata de una función **total**. Si la regla omitiera a uno o más elementos del dominio, diríamos que es una función **parcial**.

Ejemplo 4 La función parcial *signo2* indefinida cuando $x = 0$:

$$\text{signo2}(x) = \begin{cases} \text{neg} & \text{si } x < 0 \\ \text{pos} & \text{si } x > 0 \end{cases}$$

En los lenguajes funcionales **fuertemente tipificados**, como Ocaml [14, 4, 3, 21], el dominio y rango de una función debe especificarse, ya sea explícitamente o bien mediante un sistema de **inferencia de tipos**. En los lenguajes no tipificados, como Lisp [18, 6, 7, 22], esto no es necesario. En el siguiente ejemplo definimos *add/2* en Ocaml, una función que suma sus dos argumentos:

```
# let add x y = x + y ;;
val add : int -> int -> int = <fun>
```

la línea 2 nos dice que *add/2* es una función que va de los enteros (*int*), a los enteros y a los enteros. Esto es, que dados dos enteros, computará un tercero. ¿Saben ustedes qué hace si sólo se le da un entero? Ocaml tiene predefinidos una serie de **tipos de datos** primitivos, que incluyen a los enteros. Como es costumbre en nuestras notas de curso, las negritas representan palabras reservadas del lenguaje de programación.

En Lisp, la definición de *add* sería como sigue:

```
CL-USER > (defun add (x y) (+ x y))
ADD
```

la línea 2 nos dice que *ADD* ha sido definida como función. Este curso les ofrecerá los elementos necesarios para decidir cuando les conviene usar un lenguaje fuertemente tipificado y cuando no.

Si queremos definir *signo/1* en Ocaml, antes debemos definir un tipo de datos que represente al rango de la función $\{\text{pos}, \text{cero}, \text{neg}\}$:

```
1 # type signos = Neg | Cero | Pos ;;
2 type signos = Neg | Cero | Pos
3 # let signo x =
4     if x<0 then Neg else
5     if x=0 then Cero else Pos ;;
6 val signo : int -> signos = <fun>
7 # signo 5 ;;
8 - : signos = Pos
9 # signo 0 ;;
10 - : signos = Cero
11 # signo (-2) ;;
12 - : signos = Neg
```

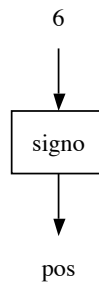
Observen que la función *signo*/1 está definida como un mapeo de enteros a signos (línea 6). Los paréntesis en la línea 11 son necesarios, pues `'-/1` es una función y queremos aplicar *signo*/1 a `-2` y no a `-`.

En Lisp, la definición de *signo* no requiere de una declaración de tipo:

```
CL-USER> (defun signo (x)
           (cond ((< x 0) 'neg)
                 ((zerop x) 'cero)
                 (t 'pos)))
SIGNO
CL-USER> (signo 3)
POS
CL-USER> (signo -2)
NEG
CL-USER> (signo 0)
CERO
```

Podemos ver a una función como una caja negra con entradas representadas por sus parámetros y una salida representando el resultado computado por la función. La salida obviamente es uno de los valores del rango de la función en cuestión. La elección de qué valor se coloca en la salida está determinada por la regla que define a la función.

Ejemplo 5 *La función signo como caja negra:*



6 aquí se conoce como **parámetro actual** de la función, es decir el valor que se le provee a la función. Al proceso de proveer un parámetro actual a una función se le conoce como **aplicación** de la función. En el ejemplo anterior diríamos que *signo* se aplica a 6, para expresar que la regla de *signo* es invocada usando 6 como parámetro actual. En muchas ocasiones nos referiremos al parámetro actual y formal de una función como los **argumentos** de la función. La aplicación de la función *signo* a 6 puede expresarse en notación matemática:

$$\textit{signo}(6)$$

Decimos que esta aplicación *evaluó* a *pos*, lo que escribimos como:

$$\textit{signo}(6) \rightarrow \textit{pos}$$

La expresión anterior también indica que *pos* es la salida de la caja negra *signo* cuando se le provee el parámetro actual 6.

La idea de una función como una transformadora de entradas en salidas es uno de los fundamentos de la programación funcional. Las cajas negras proveen bloques de construcción para un programa funcional, y uniendo varias cajas es posible especificar operaciones más sofisticadas. El proceso de “ensamblar cajas” se conoce como **composición** de funciones.

Para ilustrar este proceso de composición, definimos a continuación la función *max* que computa el máximo de un par de números *m* y *n*

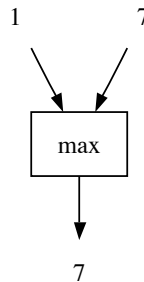
$$\text{max}(m,n) = \begin{cases} m & \text{si } m > n \\ n & \text{en cualquier otro caso} \end{cases}$$

El dominio de *max* es el conjunto de pares de números enteros y el rango es el conjunto de los enteros. Esto se puede escribir en notación matemática como:

$$\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$$

Podemos ver a *max* como una caja negra para computar el máximo de dos números:

Ejemplo 6 *La función max como caja negra:*



Lo cual escribimos:

$$\text{max}(1,7) \rightarrow 7$$

En Ocaml, nuestra función *max/1* quedaría definida como:

```

1 | # let max (m,n) = if m > n then m else n ;;
2 | val max : 'a * 'a -> 'a = <fun>
3 | # max(1,7) ;;
4 | - : int = 7

```

La línea 2 nos dice que *max/1* es una **función polimórfica**, es decir, que acepta argumentos de varios tipos. Lo mismo puede computar el máximo de un par de enteros, que de un par de reales. Observan también que la aridad de la función es

uno, pues acepta como argumento un par de valores numéricos de cualquier tipo. El resultado computado es del mismo tipo que los valores numéricos de entrada. Si queremos estrictamente una función de pares de enteros a enteros podemos usar:

```

1 # let max (m:int), (n:int) = if m > n then m else n ;;
2 val max : int * int -> int = <fun>
3 # max(3.4,5.6) ;;
4 Characters 3-12:
5   max(3.4,5.6) ;;
6   ^^^^^^^^^
7 Error: This expression has type float * float but is here used
8       with type int * int

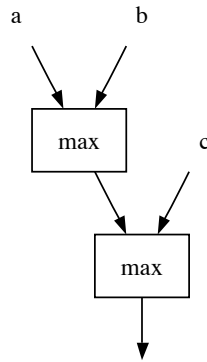
```

Ahora podemos usar *max* como bloque de construcción para obtener funciones más complejas. Supongamos que requerimos de una función que computa el máximo de tres números, en lugar de sólo dos. Podemos definir esta función como *max3*:

$$\max3(a,b,c) = \begin{cases} a & \text{si } a \geq b \text{ y } a > c \text{ o } a \geq c \text{ y } a > b \\ b & \text{si } b \geq a \text{ y } b > c \text{ o } b \geq c \text{ y } b > a \\ c & \text{si } c \geq a \text{ y } c > b \text{ o } c \geq b \text{ y } c > a \\ a & \text{en cualquier otro caso} \end{cases}$$

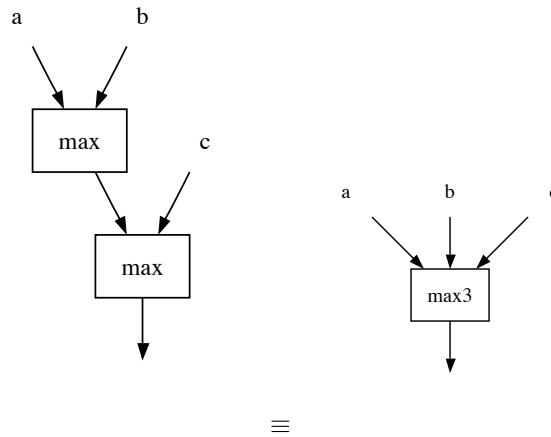
El último caso se requiere cuando $a = b = c$. Esta definición es bastante complicada. Una forma mucho más elegante de definir *max3* consiste en usar la función *max* previamente definida.

Ejemplo 7 La función *max3* como composición usando *max*:



Lo cual escribimos como $\max3(a,b,c) = \max(\max(a,b),c)$. Podemos tratar a *max3* como una caja negra con tres entradas.

Ejemplo 8 La función *max3* como caja negra usando la composición de *max*:

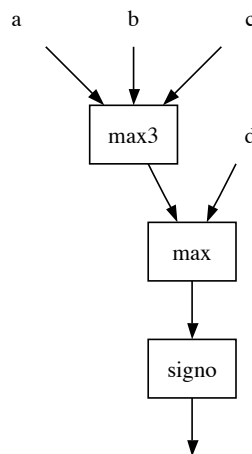


En Ocaml, la función *max3*/1 se escribiría como:

```
1 | # let max3 (a,b,c) = max(max (a,b), c) ;;
2 | val max3 : int * int * int -> int = <fun>
```

Ahora podemos olvidarnos de los detalles internos de *max3* y usar esta función como una caja negra para construir nuevas funciones.

Ejemplo 9 La función *signomax4* computa el signo del máximo de 4 números:



Que escribimos como $signomax4(a,b,c,d) = signo(max(max3(a,b,c),d))$. En Ocaml se definiría como:

```

1 | # let signomax4(a,b,c,d) = signo(max(max3(a,b,c),d)) ;;
2 | val signomax4 : int * int * int * int -> signos = <fun>

```

Así que, dados un conjunto de funciones **predefinidas** o **primitivas**, por ejemplo aritmética básica, podemos construir nuevas funciones en términos de esas primitivas. Luego, esas nuevas funciones pueden usarse para construir nuevas funciones más complejas.

1.3. Transparencia referencial

La propiedad fundamental de las funciones matemáticas que permite la analogía con los bloques de construcción se llama **transparencia referencial**. Intuitivamente esto quiere decir el valor de una expresión, depende exclusivamente del valor de las sub expresiones que lo componen, evitando así la presencia de **efectos colaterales** propios de lenguajes que presentan **opacidad referencial**.

Una función con referencia transparente tiene como característica que dados los mismos parámetros para su aplicación, obtendremos siempre el mismo resultado. Mientras que en matemáticas todas las funciones ofrecen transparencia referencial, ese no es el caso en los lenguajes de programación. Consideren la función `GetInput ()`, su salida depende del lo que el usuario teclee! Múltiples llamadas a la función `GetInput` con el mismo parámetro (una cadena vacía), producen diferentes resultados.

Veamos otro ejemplo. Una persona evaluando la expresión $(2ax + b)(2ax + c)$ no se molestaría jamás por evaluar dos veces la sub expresión $2ax$. Una vez que determina que $2ax = 12$, la persona substituirá 12 por ambas ocurrencias de $2ax$. Esto se debe a que una expresión aritmética dada en un contexto fijo, producirá siempre el mismo valor como resultado. Dados los valores $a = 3$ y $x = 2$, $2ax$ será siempre igual a 12. La transparencia referencial resulta del hecho de que los operadores aritméticos no tienen memoria, por lo que toda llamada al operador con los mismos parámetros actuales, producirá la misma salida.

¿Porqué es importante una propiedad como la transparencia referencial? Por las matemáticas sabemos lo importante de poder substituir iguales por iguales. Esto nos permite derivar nuevas ecuaciones, a partir de las ecuaciones dadas, transformar expresiones en formas más útiles y probar propiedades acerca de tales expresiones. En el contexto de los lenguajes de programación, la transparencia referencial permite además optimizaciones tales como la eliminación de subexpresiones comunes, como en el ejemplo anterior $2ax$.

Observemos ahora que pasa con lenguajes que no ofrecen transparencia referencial. Consideren la siguiente definición de una pseudofunción en Pascal:

```

1 | function F (x:integer) : integer;
2 |   begin
3 |     a := a+1;
4 |     F := x*x;
5 |   end

```

Debido a que F guarda un registro en a del número de veces que la función ha sido aplicada, no podríamos eliminar la subexpresión común en la expresión $(a + 2 * F(b)) * (c + 2 * F(b))$. Esto debido a que al cambiar el número de veces que F ha sido aplicada, cambia el resultado de la expresión.

1.4. Funciones de orden superior

Otra idea importante en la programación funcional es el concepto de **función de orden superior**, es decir, funciones que toman otras funciones como sus argumentos, o bien, regresan funciones como su resultado. La derivada y la antiderivada en el cálculo, son ejemplos de funciones que mapean a otras funciones.

Las funciones de orden superior permiten utilizar la **técnica de Curry** en la cual una función es aplicada a sus argumentos, uno a la vez. Cada aplicación regresa una función de orden superior que acepta el siguiente argumento. He aquí un ejemplo en Ocaml para la función *suma/2* en dos versiones. La primera de ellas enfatiza el carácter curry del ejemplo:

```

1 | # let suma = function x -> function y -> x+y ;;
2 | val suma : int -> int -> int = <fun>
3 | # let suma x y = x + y ;;
4 | val suma : int -> int -> int = <fun>
5 | # suma 3 4 ;;
6 | - : int = 7
7 | # suma 3 ;;
8 | - : int -> int = <fun>

```

El tipo de *suma/2* nos indica que esta función toma sus argumentos uno a uno. Puede ser aplicada a dos argumentos, como suele hacerse normalmente (líneas 5 y 6); pero puede ser llamada con un sólo argumento, ¡regresando una función en el dominio de enteros a enteros! (líneas 7 y 8). Esta aplicación espera un segundo argumento para dar el resultado de la suma.

Las funciones de orden superior pueden verse como un nuevo pegamento conceptual que permite usar funciones simples para definir funciones más complejas. Esto se puede ilustrar con un problema de procesamiento de listas: la suma de los miembros de una lista. Recuerden que una lista es un **tipo de dato recursivo**, la lista es una lista vacía (*nil*) o algo pegado (*cons*) a una lista:

```

1 | listade X ::= nil | cons X (listade X)

```

Por ejemplo: *nil* es la lista vacía; a veces la lista vacía también se representa como []; [1] es una abreviatura de *cons 1 nil*; y [1,2,3] es una abreviatura de *cons 1 (cons 2 (cons 3 nil))*.

La *sumatoria* de los elementos de una lista se puede computar con una función recursiva:

$$\begin{aligned} \text{sumatoria nil} &= 0 \\ \text{sumatoria (cons num lst)} &= \text{num} + \text{sumatoria lst} \end{aligned}$$

Si pensamos en cómo programar el *producto* de los miembros de una lista, observaremos que esa operación y la *sumatoria* que acabamos de definir, pueden plantearse como un patrón recursivo recurrente general, conocido como *reduce*:

$$\text{sumatoria} = \text{reduce suma 0}$$

donde por conveniencia, en lugar de un operador infijo $+$ para la suma, usamos la función *suma/2* definida previamente:

$$\text{suma } x \ y = x + y$$

La definición de *reduce/3* es la siguiente:

$$\begin{aligned} \text{reduce } f \ x \ \text{nil} &= x \\ \text{reduce } f \ x \ (\text{cons } a \ l) &= (f \ a) (\text{reduce } f \ x \ l) \end{aligned}$$

Observen que al definir *sumatoria/1* estamos usando *reduce/3* con sólo dos argumentos, por lo que el llamado resulta en una función del argumento restante. En general, una función de aridad n , aplicada a sólo $m < n$ argumentos, genera una función de aridad $n - m$. En el resto del artículo ejemplificare los conceptos con código en Ocaml. Observen la definición de *suma/2* y las dos llamadas a esta función:

```

1 # let suma x y = x + y;;
2 val suma : int -> int -> int = <fun>
3 # suma 4 5;;
4 - : int = 9
5 # suma 4;;
6 - : int -> int = <fun>
```

la definición de *suma/2* (línea 1) nos dice que *suma* es una función de los enteros, a los enteros, a los enteros (línea 2); esto es, el resultado de computar *suma* es también un entero, como puede observarse en la primera aplicación de la función (línea 3) cuyo resultado es el entero nueve (línea 4). Sin embargo, la segunda llamada a la función (línea 5), da como resultado otra función de los enteros a los enteros (línea 6), esto es, una función del argumento restante. Funciones como la obtenida en esta última llamada reciben el nombre de *curryficadas*. Veamos la definición de *reduce/3* y *sumatoria/1*:

```

1 # let rec reduce f x l = match l with
2   [] -> x |
3   h::t -> f h (reduce f x t) ;;
4 val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
5 # let sumatoria = reduce suma 0 ;;
6 val sumatoria : int list -> int = <fun>
```

la definición de *reduce* es recurrente, de ahí que se incluya la palabra reservada *rec* (línea 1) para que el nombre de la función pueda usarse en su misma definición al hacer el llamado recurrente. Esta función recibe tres argumentos una función *f*, un elemento *x* y una lista de elementos *l*. Si la lista de elementos está vacía, la función regresa *x*; si ese no es el caso, aplica *f* al primero elemento de la lista (línea 2), y esa función curryficada es aplicada al *reduce* del resto de la lista (línea 3). La signatura de *reduce/3* (línea 4) nos indica que hemos definido una **función de orden superior**, es decir, una función que recibe funciones como argumento. La lectura de los tipos aquí es como sigue: *l* es una lista de elementos de tipo '*a*' (cualquiera que sea el caso); *x* es de tipo '*b*'; puesto que *f* se aplica a elementos de la lista y en última instancia a *x*, se trata de una función de '*a*' a '*b*' a '*b*'; y por tanto el resultado de la función *reduce* es de tipo '*b*'. La función *suma/2* recibe una lista de enteros y produce como resultado un entero. La inferencia de tipos en este caso es porque hemos introducido 0 en la llamada y Ocaml puede inferir que '*b*' es este caso es el conjunto de los enteros.

Ahora viene lo interesante, podemos definir el producto de una lista reutilizando *reduce*:

```

1 # let mult x y = x * y ;;
2 val mult : int -> int -> int = <fun>
3 # let producto = reduce mult 1;;
4 val producto : int list -> int = <fun>
5 # producto [1;2;3;4];;
6 - : int = 24
7 # suma [1;2;3;4];;
8 - : int = 10

```

Una forma intuitiva de entender *reduce* es considerarla como un transformador de listas que substituye cada *cons* por el valor de su primer argumento *f* y cada *nil* por el valor del segundo *x*. Así, la llamada a *producto*[1;2;3] es en realidad una abreviatura de

$$\text{producto cons 1 (cons 2 (cons 3 []))}$$

que *reduce* convierte a

$$\text{mult 1 (mult 2 (mult 3 1))}$$

lo cual evalúa a 6.

Veamos otro ejemplo sobre este patrón recurrente aplicado a listas. Podemos hacer explícito el operador *cons* con la siguiente definición:

```

1 # let cons x y = x :: y;;
2 val cons : 'a -> 'a list -> 'a list = <fun>
3 # cons 1 [] ;;
4 - : int list = [1]
5 # cons 1 (cons 2 []) ;;
6 - : int list = [1; 2]

```

cuya signatura nos dice que *x* debe ser un elemento de un cierto tipo, y que *y* es una lista de elementos de ese mismo tipo. La salida de la función es la lista construida al

agregar x al frente de y , como lo muestran los ejemplos a partir de la línea 3. Ahora que contamos con `cons/2` podemos definir `append/2` usando esta definición:

```
1 # let append lst1 lst2 = reduce cons lst2 lst1;;
2 val append : 'a list -> 'a list -> 'a list = <fun>
3 # append [1;2] [3;4] ;;
4 - : int list = [1; 2; 3; 4]
```

O, siguiendo la misma estrategia, podemos definir una función que reciba una lista enteros y regrese la lista formada por el doble de los enteros originales:

```
1 # let dobleycons num lst = cons (2*num) lst;;
2 val dobleycons : int -> int list -> int list = <fun>
3 # let dobles = reduce dobleycons [];;
4 val dobles : int list -> int list = <fun>
5 # dobles [1;2;3;4];;
6 - : int list = [2; 4; 6; 8]
```

Las funciones de orden superior nos permiten definir fácilmente la **composición funcional** `o` y el **mapeo** sobre listas `map`:

```
1 # let o f g h = f (g h);;
2 val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
3 # let map f = reduce (o cons f) [];;
4 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

De forma que ahora podemos definir `dobles2/1` en términos de una **función anónima** y un mapeo:

```
1 # let dobles2 l = map (fun x -> 2*x) l;;
2 val dobles2 : int list -> int list = <fun>
3 # dobles2 [1;2;3;4] ;;
4 - : int list = [2; 4; 6; 8]
```

o bien, podemos extender nuestro concepto de sumatoria para que trabaje sobre matrices representadas como listas de listas:

```
1 # let sumatoria_matriz = o sumatoria (map sumatoria);;
2 val sumatoria_matriz : int list list -> int = <fun>
3 # sumatoria_matriz [[1;2];[3;4]];;
4 - : int = 10
```

1.5. Recursividad

Las iteraciones de la programación tradicional, son normalmente implementados de manera recursiva en la programación funcional. Las **funciones recursivas** [15], como hemos visto, se definen en términos de ellas mismas, permitiendo de esta forma que una operación se repita una y otra vez. La recursión a la cola permite

optimizar la implementación de estas funciones. La razón por la cual las funciones recursivas son naturales en los lenguajes funcionales, es porque normalmente en ellos operamos con estructuras de datos (tipos de datos) recursivas. Aunque las listas están definidas en estos lenguajes, observen las siguientes definiciones de tipo “lista de” y “árbol de” en Ocaml:

```

1 # type 'a lista = Nil | Cons of 'a * 'a lista;;
2 type 'a lista = Nil | Cons of 'a * 'a lista
3 # type 'a arbolbin = Hoja
4 | Nodo of 'a * 'a arbolbin * 'a arbolbin;;
5 type 'a arbolbin = Hoja | Nodo of 'a * 'a arbolbin * 'a arbolbin

```

son definiciones de tipos de datos !recursivas! Una lista vacía es una lista y algo pegado a una lista vacía es una lista. Una hoja es un árbol, y un nodo pegado a dos árboles, es un árbol. A continuación definiremos *miembro/2* para estos dos tipos de datos:

```

1 # let rec miembro elt lst = match lst with
2 | Nil -> false
3 | Cons(e,l) -> if e=elt then true else miembro elt l;;
4 val miembro : 'a -> 'a lista -> bool = <fun>
5 # let rec miembro elt arbol = match arbol with
6 | Hoja -> false
7 | Nodo(e,izq,der) -> if e=elt then true
8 | else miembro elt izq || miembro elt der;;
9 val miembro : 'a -> 'a arbolbin -> bool = <fun>
10 # miembro 2 (Cons(1,Nil)) ;;
11 - : bool = false
12 # miembro 1 (Cons(1,Nil)) ;;
13 - : bool = true
14 # miembro 2 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja))) ;;
15 - : bool = true
16 # miembro 4 (Nodo(1,Nodo(2,Hoja,Hoja),Nodo(3,Hoja,Hoja))) ;;
17 - : bool = false

```

Los patrones y la recursividad pueden factorizarse utilizando funciones de orden superior, los catamorfismos y los anamorfismos son los ejemplos más obvios. Estas funciones de orden superior juegan un papel análogo a las estructuras de control de la programación imperativa.

1.6. Consideraciones

Hemos presentado las bondades de la programación funcional para responder a la pregunta de porqué es relevante estudiar este paradigma de programación. Una postura similar para responder a esta pregunta puede encontrarse en el artículo de Hugues [10] *Why Fuctional Programming matters*. Hudak [9] nos ofrece otro artículo introductorio interesante, por la perspectiva histórica que asume, al presentar los lenguajes funcionales de programación. Dos textos donde pueden revisarse los conceptos generales de la programación funcional, son el libro de Field y Harrison [5] y el de MacLennan [13].

Parte I
Lisp

Capítulo 2

Introducción a Lisp

Resumen El objetivo de este capítulo es que puedan programar en Lisp tan pronto como sea posible. Al final de la misma, conocerán lo suficiente de Lisp como para comenzar a escribir sus propios programas. Este material debe revisarse como un tutorial básico sobre el lenguaje Lisp.

2.1. Expresiones

Es particularmente cierto que la mejor forma de aprender Lisp es usándolo, porque se trata de un lenguaje interactivo. Cualquier sistema Lisp, incluye una interfaz interactiva llamada *top-level*. Uno escribe expresiones Lisp en el *top-level*, y el sistema despliega sus valores. El sistema normalmente despliega un indicador (el *prompt* `>`) de que está esperando que una expresión sea escrita. Ej. Una de las expresiones Lisp más simples es el entero, si escribimos el entero 1 después del *prompt* y tecleamos *enter*, tenemos:

```
> 1
1
>
```

el sistema despliega el valor de la expresión, seguida de un nuevo *prompt*, indicando que está listo para evaluar una nueva expresión. En este caso, el sistema desplegó lo mismo que tecleamos porque los números evalúan a si mismos. Las cosas son más interesantes cuando una expresión necesita algo de trabajo para ser evaluado. Ej. Sumar dos números:

```
> (+ 2 3)
5
>
```

En la expresión `(+ 2 3)` el `+` es llamado el *operador* y los números 2 y 3 son sus *argumentos*. Como el operador viene al principio de la expresión, esta notación

se conoce como *prefija* y aunque parezca extraña al principio, veremos que es uno de los mejores aspectos de Lisp. Por ejemplo, si queremos sumar tres números en notación “normal”, necesitaríamos usar dos veces el operador +: 2+3+4. En Lisp sólo necesitamos agregar otro argumento. Ej. Las siguientes sumas son válidas en Lisp:

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 5)
10
```

Como los operadores pueden tomar un número variable de argumentos, es necesario utilizar los paréntesis para indicar donde inicia y donde termina una expresión. Las expresiones pueden anidarse, esto es, el argumento de una expresión puede ser otra expresión compleja. Ej.

```
> (/ (- 7 1) (- 4 2))
3
```

En español esto corresponde a siete menos uno, dividido por cuatro menos dos.

Estética minimalista, esto es todo lo que hay que decir sobre la notación en Lisp. Toda expresión Lisp es un *átomo*, como 1, o bien es una *lista* que consiste de cero o más expresiones delimitadas por paréntesis. Como veremos, código y datos usan esta notación en Lisp.

2.2. Evaluación

Veamos más en detalle como las expresiones son evaluadas para desplegar su valor en el top-level. En Lisp, + es una función y (+ 2 3) es una *llamada* a la función. Cuando Lisp evalúa una llamada a alguna función, lo hace en dos pasos:

1. Los argumentos de la llamada son evaluados de izquierda a derecha. En este caso, los valores de los argumentos son 2 y 3, respectivamente.
2. Los valores de los argumentos son pasados a la función nombrada por el operador. En este caso la función + que regresa 5.

Si alguno de los argumentos es a su vez una llamada de función, será evaluado con las mismas reglas. Ej. Al evaluar la expresión (/ (- 7 1) (- 4 2)) pasa lo siguiente.

1. Lisp evalúa el primer argumento de izquierda a derecha (- 7 1). 7 es evaluado como 7 y 1 como 1. Estos valores son pasados a la función - que regresa 6.
2. El siguiente argumento (- 4 2) es evaluado. 4 es evaluado como 4 y 2 como 2. Estos valores son pasados a la función - que regresa 2.

3. Los valores 6 y 2 son pasados a la función / que regresa 3.

No todos los operadores en Lisp son funciones, pero la mayoría lo son. Todas las llamadas a función son evaluadas de esta forma, que se conoce como *regla de evaluación* de Lisp.

Un operador Lisp que no sigue la regla de evaluación es `quote` (se abrevia `'`). Los operadores que no siguen la regla de evaluación se conocen como *operadores especiales*. La regla de evaluación de `quote` es –No hagas nada, despliega lo que el usuario tecleo, verbatim. Ej.

```
> (quote (+ 2 3))
(+ 2 3)
> '(+ 2 3)
(+ 2 3)
```

Lisp provee el operador `quote` como una forma de evitar que una expresión sea evaluada. En la siguiente sección veremos porque esta protección puede ser útil.

2.3. Datos

Lisp ofrece los tipos de datos que podemos encontrar en otros lenguajes de programación y otros que no. Un tipo de datos que ya usamos es el *entero*, que se escribe como una secuencia de dígitos. Ej. 256. Otro tipo de datos es la *cadena* de caracteres que se delimita por comillas. Ej., “ora et labora, pos ora”. Enteros y cadenas evalúan a ellos mismos.

Dos tipos de datos propios de Lisp son los *símbolos* y las *listas*. Los símbolos son palabras. Normalmente se evalúan como si estuvieran escritos en mayúsculas, independientemente de como fueron tecleados. Ej.

```
> 'Amarone
AMARONE
```

Los símbolos por lo general no evalúan a sí mismos, así que si es necesario referirse a ellos, se debe usar `quote`, como en ejemplo anterior.

Las listas se representan como cero o más elementos entre paréntesis. Los elementos pueden ser de cualquier tipo, incluidas las listas. Se debe usar `quote` con las listas, ya que de otra forma Lisp las tomaría como una llamada a función. Ej.

```
> '(Mis 2 "ciudades")
(MIS 2 "CIUDADES")
> '(La lista (a b c) tiene 3 elementos)
(LA LISTA (A B C) TIENE 3 ELEMENTOS)
```

Observen que un sólo `quote` protege a toda la expresión, incluidas las expresiones en ella.

Se puede construir listas usando el operador `list` que es una función, y por lo tanto, sus argumentos son evaluados. Ej.

```
> (list 'mis (+ 4 2) "colegas")
(MIS 6 COLEGAS)
```

Estética minimalista y pragmática, observen que los programas Lisp se representan como listas. Si el argumento estético no bastará para defender la notación de Lisp, esto debe bastar –Un programa Lisp puede generar código Lisp! Por eso es necesario `quote`. Si una lista es precedida por el operador `quote`, la evaluación regresa la misma lista, en otro caso, la lista es evaluada como si fuese código. Ej.

```
> (list '(+ 2 3) (+ 2 3))
((+ 2 3) 5)
```

En Lisp hay dos formas de representar la lista vacía, como un par de paréntesis o con el símbolo `NIL`. Ej.

```
> ()
NIL
> NIL
NIL
```

2.4. Operaciones con listas

La función `cons` construye listas. Si su segundo argumento es una lista, regresa una nueva lista con el primer argumento agregado en el frente. Ej.

```
> (cons 'a '(b c d))
(A B C D)
> (cons 'a (cons 'b nil))
(A B)
```

El segundo ejemplo es equivalente a:

```
(A B)
```

Las funciones primitivas para acceder los elementos de una lista son `car` y `cdr`. El `car` de una lista es su primer elemento (el más a la izquierda) y el `cdr` es el resto de la lista (menos el primer elemento). Ej.

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

Se pueden usar combinaciones de `car` y `cdr` para acceder cualquier elemento de la lista. Ej.

```
> (car (cdr (cdr '(a b c d))))
C
> (caddr '(a b c d))
C
> (third '(a b c d))
C
```

2.5. Valores de verdad

En Lisp, el símbolo `t` es la representación por default para verdadero. La representación por default de falso es `nil`. Ambos evalúan a sí mismos. Ej. La función `listp` regresa verdadero si su argumento es una lista:

```
> (listp '(a b c))
T
> (listp 34)
NIL
```

Una función cuyo valor de regrese se interpreta como un valor de verdad (verdadero o falso) se conoce como *predicado*. En Lisp es común que el símbolo de un predicado termine en `p`.

Como `nil` juega dos roles en Lisp, las funciones `null` (lista vacía) y `not` (negación) hacen exactamente lo mismo:

```
> (null nil)
T
> (not nil)
T
```

El condicional (estructura de control) más simple en Lisp es `if`. Normalmente toma tres argumentos: una expresión *test*, una expresión *then* y una expresión *else*. La expresión *test* es evaluada, si su valor es verdadero, la expresión *then* es evaluada; si su valor es falso, la expresión *else* es evaluada. Ej.

```
> (if (listp '(a b c d))
      (+ 1 2)
      (+ 3 4))
3
> (if (listp 34)
      (+ 1 2)
      (+ 3 4))
7
```

Como `quote`, `if` es un operador especial. No puede implementarse como una función, porque los argumentos de una función siempre se evalúan, y la idea al usar `if` es que sólo uno de sus argumentos sea evaluado.

Si bien el default para representar verdadero es `t`, todo excepto `nil` cuenta como verdadero en un contexto lógico. Ej.

```
> (if 27 1 2)
1
> (if nil 1 2)
2
```

Los operadores lógicos `and` y `or` parecen condicionales. Ambos toman cualquier número de argumentos, pero solo evalúan los necesarios para decidir que valor regresar. Si todos los argumentos son verdaderos (diferentes de `nil`), entonces `and` regresa el valor del último argumento. Ej.

```
> (and t (+ 1 2))
3
```

Pero si uno de los argumentos de `and` resulta falso, ninguno de los argumentos posteriores es evaluado y el operador regresa `nil`. De manera similar, `or` se detiene en cuanto encuentra un elemento verdadero.

```
> (or nil nil (+ 1 2) nil)
3
```

Observen que los operadores lógicos son operadores especiales, en este caso definidos como *macros*.

2.6. Funciones

Es posible definir nuevas funciones con `defun` que toma normalmente tres argumentos: un nombre, una lista de parámetros y una o más expresiones que conforman el cuerpo de la función. Ej. Así definiríamos `third`:

```
> (defun tercero (lst)
  (caddr lst))
TERCERO
```

El primer argumento de `defun` indica que el nombre de nuestra función definida será `tercero`. El segundo argumento `(lst)` indica que la función tiene un sólo argumento, `lst`. Un símbolo usado de esta forma se conoce como *variable*. Cuando la variable representa el argumento de una función, se conoce como *parámetro*. El resto de la definición indica lo que se debe hacer para calcular el valor de la función, en este caso, para cualquier `lst`, se calculará el primer elemento, del resto, del resto del parámetro `(caddr lst)`. Ej.

```
> (tercero '(a b c d e))
c
```

Ahora que hemos introducido el concepto de variable, es más sencillo entender lo que es un símbolo. Los símbolos son nombres de variables, que existen con derechos

propios en el lenguaje Lisp. Por ello símbolos y listas deben protegerse con `quote` para ser accesados. Una lista debe protegerse porque de otra forma es procesada como si fuese código; un símbolo debe protegerse porque de otra forma es procesado como si fuese una variable.

Podríamos decir que la definición de una función corresponde a la versión generalizada de una expresión Lisp. Ej. La siguiente expresión verifica si la suma de 1 y 4 es mayor que 3:

```
> (> (+ 1 4) 3)
T
```

Substituyendo los números particulares por variables, podemos definir una función que verifica si la suma de sus dos primeros argumentos es mayor que el tercero:

```
> (defun suma-mayor-que (x y z)
  (> (+ x y) z))
SUMA-MAYOR-QUE
> (suma-mayor-que 1 4 3)
T
```

Lisp no distingue entre programa, procedimiento y función; todos cuentan como funciones y de hecho, casi todo el lenguaje está compuesto de funciones. Si se desea considerar una función en particular como *main*, es posible hacerlo, pero cualquier función puede ser llamada desde el top-level. Entre otras cosas, esto significa que es posible probar nuestros programas, pieza por pieza, conforme los vamos escribiendo, lo que se conoce como *programación incremental (bottom-up)*.

2.7. Recursividad

Las funciones que hemos definido hasta ahora, llaman a otras funciones para hacer una parte de sus cálculos. Ej. `suma-mayor-que` llama a las funciones `+` y `>`. Una función puede llamar a cualquier otra función, incluida ella misma.

Una función que se llama a sí misma se conoce como *recursiva*. Ej. En Lisp la función `member` verifica cuando algo es miembro de una lista. He aquí una versión recursiva simplificada de esta función:

```
> (defun miembro (obj lst)
  (if (null lst)
      nil
      (if (eql (car lst) obj)
          lst
          (miembro obj (cdr lst)))))
MIEMBRO
```

El predicado `eql` verifica si sus dos argumentos son idénticos, el resto lo hemos visto previamente. La llamada a `miembro` es como sigue:

```
> (miembro 'b '(a b c))
(B C)
> (miembro 'z '(a b c))
NIL
```

La descripción en español de lo que hace la función `miembro` es como sigue:

1. Primero, verificar si la lista `lst` está vacía, en cuyo caso es evidente que `obj` no es un miembro de `lst`.
2. De otra forma, si `obj` es el primer elemento de `lst` entonces es miembro de la lista.
3. De otra forma, `obj` es miembro de `lst` únicamente si es miembro del resto de `lst`.

Traducir una función recursiva a una descripción como la anterior, siempre ayuda a entenderla. Al principio, es común toparse con dificultades para entender la recursividad. En gran medida esto se debe a que usamos la metáfora equivocada de función. Tenemos la tendencia de pensar en una función como cierta forma de máquina, donde las materias primas llegan como parámetros. Parte del trabajo se da por encargo a otras funciones y el resultado final se empaqueta y se expide como el valor de la función. Si se usa esta metáfora la recursividad resulta paradójica – ¿Qué sentido tiene darse por encargo trabajo cuando uno ya está ocupado?

Una metáfora más adecuada es ver a las funciones como procesos que vamos resolviendo. Solemos usar procesos recursivos en nuestras actividades diarias. Por ejemplo, supongan a un historiador interesado en los cambios de población a través de la historia de Europa. El proceso que el historiador utilizaría para examinar un documento es el siguiente:

1. Obtener una copia del documento que le interesa.
2. Buscar en él la información relativa a cambios de población en Europa.
3. Si el documento menciona otros documentos que puede ser útiles, examinarlos.

Piensen en `miembro` como las reglas que definen cuando algo es miembro de una lista y no como una máquina que computa si algo es miembro de una lista. De esta forma, la paradoja desaparece.

2.8. Leyendo y escribiendo Lisp

Estética. Si bien los paréntesis delimitan las expresiones en Lisp, un programador en realidad usa los márgenes en el código para hacerlo más legible. Casi todo editor puede configurarse para verificar paréntesis bien balanceados. Ej. `:set sm` en el editor `vi`; o `M-x lisp-mode` en `Emacs`. Cualquier hacker en Lisp tendría problemas para leer algo como:

```
> (defun miembro (obj lst) (if (null lst) nil (if
```

```
(eql (car lst) obj) lst (miembro obj (cdr lst))))
MIEMBRO
```

2.9. Entradas y salidas

Hasta el momento hemos procesado las E/S implícitamente, utilizando el top-level. Para programas interactivos esto no es suficiente¹, así que veremos algunas operaciones básicas de E/S.

La función de salida más general en Lisp es `format`. Esta función toma dos o más argumentos: el primero indica donde debe imprimirse la salida, el segundo es una cadena que se usa como molde (*template*), y el resto son generalmente objetos cuya representación impresa será insertada en la cadena molde. Ej.

```
> (format t "~A mas ~A igual a ~A. ~%" 2 3 (+ 2 3))
2 MAS 3 IGUAL A 5.
NIL
```

Observen que dos líneas fueron desplegadas en el ejemplo anterior. La primera es producida por `format` y la segunda es el valor devuelto por la llamada a `format`, desplegada por el top-level como se hace con toda función. Normalmente no llamamos a `format` en el top-level, sino dentro de alguna función, por lo que el valor que regresa queda generalmente oculto.

El primer argumento de `format`, `t`, indica que la salida será desplegada en el dispositivo estándar de salida, generalmente el top-level. El segundo argumento es una cadena que sirve como molde de lo que será impreso. Dentro de esta cadena, cada `~A` reserva espacio para insertar un objeto y el `%` indica un salto de línea. Los espacios reservados de esta forma, son ocupados por el resto de los argumentos en el orden en que son evaluados.

La función estándar de entrada es `read`. Sin argumentos, normalmente la lectura se hace a partir del top-level. Ej. Una función que despliega un mensaje y lee la respuesta el usuario:

```
> (defun pregunta (string)
  (format t "~A" string)
  (read))
PREGUNTA
> (pregunta "Su edad: ")
Su edad: 34
34
```

Puesto que `read` espera indefinidamente a que el usuario escriba algo, no es recomendable usar `read` sin desplegar antes un mensaje que solicite la información al usuario. De otra forma el sistema dará la impresión de haberse plantado.

¹ De hecho, casi todo el software actual incluye alguna interfaz gráfica con un sistema de ventaneo, por ejemplo, en lisp: CLIM, Common Graphics, etc.

Se debe mencionar que `read` hace mucho más que leer caracteres, es un auténtico *parser* de Lisp que evalúa su entrada y regresa los objetos que se hallan generados.

La función `pregunta`, aunque corta, muestra algo que no habíamos visto antes: su cuerpo incluye más de una expresión Lisp. El cuerpo de una función puede incluir cualquier número de expresiones. Cuando la función es llamada, las expresiones en su cuerpo son evaluadas en orden y la función regresará el valor de la última expresión evaluada.

Hasta el momento, lo que hemos mostrado se conoce como *Lisp puro*, esto es, Lisp sin efectos colaterales. Un efecto colateral es un cambio en el sistema Lisp producto de la evaluación de una expresión. Cuando evaluamos `(+ 2 3)`, no hay efectos colaterales, el sistema simplemente regresa el valor 5. Pero al usar `format`, además de obtener el valor `nil`, el sistema imprime algo, esto es un tipo de efecto colateral.

Cuando se escribe código sin efectos colaterales, no hay razón alguna para definir funciones cuyo cuerpo incluya más de una expresión. La última expresión evaluada en el cuerpo producirá el valor de la función, pero el valor de las expresiones evaluadas antes se perderá.

2.10. Variables

Uno de los operadores más comunes en Lisp es `let`, que permite la creación de nuevas variables locales. Ej.

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

Una expresión `let` tiene dos partes: Primero viene una lista de expresiones definiendo las nuevas variables locales, cada una de ellas con la forma (*variable expresión*). Cada variable es inicializada con el valor que regrese la expresión asociada a ella. En el ejemplo anterior se han creado dos variables, `x` e `y`, con los valores 1 y 2 respectivamente. Esas variables son válidas dentro del cuerpo de `let`.

Después de la lista de variables y valores, viene el cuerpo de `let` constituido por una serie de expresiones que son evaluadas en orden. En el ejemplo, sólo hay una llamada a `+`. Presento ahora como ejemplo una función `preguntar` más selectiva:

```
> (defun preguntar-num ()
    (format t "Por favor, escriba un numero: ")
    (let ((val (read)))
      (if (numberp val)
          val
          (preguntar-num))))
```

Esta función crea la variable local `var` para guardar el valor que regresa `read`. Como este valor puede ahora ser manipulado por Lisp, la función revisa que se ha

escrito para decidir que hacer. Si el usuario ha escrito algo que no es un número, la función vuelve a llamarse a si misma:

```
> (preguntar-num)
Por favor, escriba un numero: a
Por favor, escriba un numero: (un numero)
Por favor, escriba un numero: 3
3
```

Las variables de este estilo se conocen como `locales` porque sólo son válidas en cierto contexto. Existe otro tipo de variables llamadas `globales`, que son visibles donde sea².

Se puede crear una variable global con un símbolo dado y un valor, usando `defparameter`:

```
> (defparameter *glob* 1970)
*GLOB*
```

Esta variable es visible donde sea, salvo en contextos que definan una variable local con el mismo nombre. Para evitar errores accidentales con los nombre de las variables, se usa la convención de nombrar a las variables globales con símbolos que inicien y terminen en asterisco.

Se pueden definir también constantes globales usando `defconstant`:

```
> (defconstant limit (+ *glob* 1))
```

No hay necesidad de dar a las constantes nombres distintivos porque si se intenta usar el nombre de una constante para una variable se produce un error. Para verificar si un símbolo es el nombre de una variable global o constante, se puede usar `boundp`:

```
> (boundp '*glob*)
T
```

2.11. Asignaciones

En Lisp el operador de asignación más común es `setf`. Se puede usar para asignar valores a cualquier tipo de variable. Ej.

```
> (setf *glob* 2000)
2000
> (let ((n 10))
  (setf n 2))
```

² La distinción propia es entre variables léxicas y especiales, pero por el momento no es necesario entrar en detalles

```
n)
2
```

Cuando el primer argumento de `setf` es un símbolo que no es el nombre de una variable local, se asume que se trata de una variable global.

```
> (setf x (list 'a 'b 'c))
(A B C)
> (car x)
A
```

Esto es, es posible crear implícitamente variables globales con sólo asignarles valores. Como sea, es preferible usar explícitamente `defparameter`.

Se puede hacer más que simplemente asignar valores a variables. El primer argumento de `setf` puede ser tanto una expresión, como un nombre de variable. En el primer caso, el valor representado por el segundo argumento es insertado en el lugar al que hace referencia la expresión. Ej.

```
> (setf (car x) 'n)
N
> x
(N B C)
```

Se puede dar cualquier número de argumentos pares a `setf`. Una expresión de la forma:

```
> (setf a 'b
      c 'd
      e 'f)
F
```

es equivalente a:

```
> (set a 'b)
B
> (set b 'c)
C
> (set e 'f)
F
```

2.12. Programación funcional

La *programación funcional* significa, entre otras cosas, escribir programas que trabajan regresando valores, en lugar de modificar cosas. Es el paradigma de programación dominante en Lisp. La mayor parte de las funciones predefinidas en el lenguaje, se espera sean llamadas por el valor que producen y no por sus efectos colaterales.

La función `remove`, por ejemplo, toma un objeto y una lista y regresa una nueva lista que contiene todos los elementos de la lista original, menos el objeto indicado:

```
> (setf lst '(k a r a t e))
(K A R A T E)
> (remove 'a lst)
(K R T E)
```

¿Por qué no decir simplemente que `remove` remueve un objeto dado de una lista? Porque esto no es lo que la función hace. La lista original no es modificada:

```
> lst
(K A R A T E)
```

Si se desea que la lista original sea afectada, se puede evaluar la siguiente expresión:

```
> (setf lst (remove 'a lst))
(K R T E)
> lst
(K R T E)
```

La programación funcional significa, esencialmente, evitar `setf` y otras expresiones con el mismo tipo de efecto colateral. Esto puede parecer contra intuitivo y hasta no deseable. Si bien programar totalmente sin efectos colaterales es inconveniente, a medida que practiquen Lisp, se sorprenderán de lo poco que en realidad se necesita este tipo de efecto.

Una de las ventajas de la programación funcional es que permite la *verificación interactiva*. En código puramente funcional, se puede verificar cada función a medida que se va escribiendo. Si la función regresa los valores que esperamos, se puede confiar en que es correcta. La confianza agregada al proceder de este modo, hace una gran diferencia: un nuevo estilo de programación.

2.13. Iteración

Cuando deseamos programar algo repetitivo, algunas veces la iteración resulta más natural que la recursividad. El caso típico de iteración consiste en generar algún tipo de tabla. Ej.

```
> (defun cuadrados (inicio fin)
  (do ((i inicio (+ i 1)))
      ((> i fin) 'final)
      (format t "~A ~A ~%" i (* i i))))
CUADRADOS
> (cuadrados 2 5)
2 4
3 9
4 16
5 25
```

FINAL

La macro `do` es el operador fundamental de iteración en Lisp. Como `let`, `do` puede crear variables y su primer argumento es una lista de especificación de variables. Cada elemento de esta lista toma la forma (*variable valor-inicial actualización*). En cada iteración el valor de las variables definidas de esta forma, cambia como lo especifica la actualización.

En el ejemplo anterior, `do` crea únicamente la variable local `i`. En la primer iteración `i` tomará el valor de `inicio` y en las sucesivas iteraciones su valor se incrementará en 1.

El segundo argumento de `do` debe ser una lista que incluya una o más expresiones. La primera expresión se usa como prueba para determinar cuando debe parar la iteración. En el ejemplo, esta prueba es `(> i fin)`. El resto de la lista será evaluado en orden cuando la iteración termine. La última expresión evaluada será el valor de `do`, por lo que `cuadrados`, regresa siempre el valor `'final`.

El resto de los argumentos de `do`, constituyen el cuerpo del ciclo y serán evaluados en orden en cada iteración, donde: las variables son actualizadas, se evalúa la prueba de fin de iteración y si esta falla, se evalúa el cuerpo de `do`. Para comparar, se presenta aquí una versión recursiva de `cuadrados-rec`:

```
> (defun cuadrados-rec (inicio fin)
  (if (> inicio fin)
      'final
      (progn
        (format t "~A ~A ~%" inicio (* inicio inicio))
        (cuadrados-rec (+ inicio 1) fin))))
CUADRADOS
```

La única novedad en esta función es `progn` que toma cualquier número de expresiones como argumentos, las evalúa en orden y regresa el valor de la última expresión evaluada.

Lisp provee operadores de iteración más sencillos para casos especiales, por ejemplo, `dolist` para iterar sobre los elementos de una lista. Ej. Una función que calcula la longitud de una lista:

```
> (defun longitud (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
LONGITUD
> (longitud '(a b c))
3
```

El primer argumento de `dolist` toma la forma (*variable expresión*), el resto de los argumentos son expresiones que constituyen el cuerpo de `dolist`. Este cuerpo será evaluado con la variable instanciada con elementos sucesivos de la lista que regresa expresión. La función del ejemplo, dice – por cada `obj` en `lst`, incrementar en uno `len`.

La versión recursiva obvia de esta función `longitud` es:

```
> (defun longitud-rec (lst)
  (if (null lst)
      0
      (+ (longitud-rec (cdr lst)) 1)))
LONGITUD
```

Esta versión será menos eficiente que la iterativa porque no es recursiva a la cola (*tail-recursive*), es decir, al terminar la recursividad, la función debe seguir haciendo algo. La definición recursiva a la cola de longitud es:

```
> (defun longitud-tr (lst)
  (labels ((longaux (lst acc)
            (if (null lst)
                acc
                (longaux (cdr lst) (+ 1 acc))))))
    (longaux lst 0)))
LONGITUD-TR
```

Lo nuevo aquí es `labels` que permite definir funciones locales como `longaux`.

2.14. Funciones como objetos

En Lisp las funciones son objetos regulares como los símbolos, las cadenas y las listas. Si le damos a `function` el nombre de una función, nos regresará el objeto asociado a ese nombre. Como `quote`, `function` es un operador especial, así que no necesitamos proteger su argumento. Ej.

```
> (function +)
#<Compiled-Function + 17BA4E>
```

Esta extraño valor corresponde a la forma en que una función sería desplegada en una implementación Lisp. Hasta ahora, hemos trabajado con objetos que lucen igual cuando los escribimos y cuando Lisp los evalúa. Esto no sucede con las funciones, cuya representación interna corresponde más a un segmento de código máquina, que a la forma como la definimos.

Al igual que usamos `'` para abreviar `quote`, podemos usar `#'`, para abreviar `function`.

```
> #' +
#<Compiled-Function + 17BA4E>
```

Como sucede con otros objetos, en Lisp podemos pasar funciones como argumentos. Una función que toma una función como argumento es `apply`. Ej.

```
> (apply #' + '(1 2 3))
6
> (+ 1 2 3)
6
```

Se le puede dar cualquier número de argumentos, si se respeta que el último de ellos sea una lista. Ej.

```
> (apply #' + 1 2 '(3 4 5))
15
```

La función `funcall` hace lo mismo, pero no necesita que sus argumentos estén empaquetados en forma de lista. Ej.

```
> (funcall #' + 1 2 3)
6
```

La macro `defun` crea una función y le da un nombre, pero las funciones no tienen porque tener nombre y no necesitamos `defun` para crearlas. Como los otros tipos de objeto en Lisp, podemos definir a las funciones literalmente.

Para referirnos literalmente a un entero usamos una secuencia de dígitos; para referirnos literalmente a una función usamos una expresión *lambda* cuyo primer elemento es el símbolo `lambda`, seguido de una lista de parámetros y el cuerpo de la función. Ej.

```
> (lambda (x y)
  (x + y))
```

Una expresión `lambda` puede considerarse como el nombre de una función. Ej. Puede ser el primer elemento de una llamada de función:

```
> ((lambda (x) (+ x 100)) 1)
101
```

o usarse con `funcall`:

```
> (funcall #' (lambda (x) (+ x 100)) 1)
101
```

Entre otras cosas, esta notación nos permite usar funciones sin necesidad de nombrarlas. También podemos usar este truco para computar mapeos sobre listas:

```
CL-USER> (mapcar #' (lambda (x) (* 2 x)) '(1 2 3 4))
(2 4 6 8)
CL-USER> (defun dobles (lst)
  (mapcar #' (lambda (x) (* 2 x)) lst))
DOBLES
CL-USER> (dobles '(1 2 3 4))
(2 4 6 8)
```

2.15. Tipos

Lisp utiliza un inusual enfoque flexible sobre tipos. En muchos lenguajes, las variables tienen un tipo asociado y no es posible usar una variable sin especificar su tipo. En Lisp, los valores tienen un tipo, no las variables. Imaginen que cada objeto en Lisp tiene asociada una etiqueta que especifica su tipo. Este enfoque se conoce como *tipificación manifiesta*. No es necesario declarar el tipo de una variable porque cualquier variable puede recibir cualquier objeto de cualquier tipo. De cualquier forma, es posible definir el tipo de una variable para optimizar el código antes de la compilación.

Lisp incluye una jerarquía predefinida de subtipos y supertipos. Un objeto siempre tiene más de un tipo. Ej. el número 27 es de tipo `fixnum`, `integer`, `rational`, `real`, `number`, `atom` y `t`, en orden de generalidad incremental. El tipo `t` es el supertipo de todo tipo.

La función `typep` toma como argumentos un objeto y un especificador de tipo y regresa `t` si el objeto es de ese tipo. Ej.

```
> (typep 27 'integer)
T
```

2.16. Consideraciones

Aunque este documento presenta un bosquejo rápido de Lisp, es posible apreciar ya el retrato de un lenguaje de programación inusual. Un lenguaje con una sola sintaxis para expresar programas y datos. Esta sintaxis se basa en listas, que son a su vez objetos en Lisp. Las funciones, que son objetos del lenguaje también, se expresan como listas. Y Lisp mismo es un programa Lisp, programado casi por completo con funciones Lisp que en nada difieren a las que podemos definir.

No debe preocuparles que la relación entre todas estas ideas no sea del todo clara. Lisp introduce tal cantidad de conceptos nuevos que toma tiempo acostumbrarse a ellos y como usarlos. Solo una cosa debe estar clara: Las ideas detrás Lisp son extremadamente elegantes.

Si C es el lenguaje para escribir UNIX³, entonces podríamos describir a Lisp como el lenguaje para describir Lisp, pero eso es una historia totalmente diferente. Un lenguaje que puede ser escrito en sí mismo, es algo distinto de un lenguaje para escribir una clase particular de aplicaciones. Ofrece una nueva forma de programar: así como es posible escribir un programa en el lenguaje, ¡el lenguaje puede mejorarse para acomodarse al programa! Esto es un buen inicio para entender la esencia de Lisp.

³ Richard Gabriel

Capítulo 3

Listas en Lisp

Resumen Las listas fueron originalmente la principal estructura de datos en Lisp. De hecho, el nombre del lenguaje es un acrónimo de “LIST Processing”. Las implementaciones modernas de Lisp incluyen otras estructuras de datos. El desarrollo de programas en Lisp refleja esta historia. Las versiones iniciales del programa suelen hacer un uso intensivo de listas, que posteriormente se convierten a otros tipos de datos, más rápidos o especializados. Este capítulo muestra qué es lo que uno puede hacer con las listas y las usa para ejemplificar algunos conceptos generales de Lisp.

3.1. Conses

En el capítulo anterior se introdujeron las funciones primitivas *cons/2*, *car/1* y *cdr/1* para el manejo de listas. Lo que *cons* hace es combinar dos objetos, en uno formado de dos partes llamado **cons**. Conceptualmente, un *cons* es un par de apuntadores: el primero es el *car* y el segundo el *cdr*.

Los *conses* proveen una representación conveniente para cualquier tipo de pares. Los dos apuntadores del *cons* pueden dirigirse a cualquier objeto, incluyendo otros *conses*. Es esta última propiedad, la que explotamos para definir **listas** en términos de *cons*. Una lista puede definirse como el par formado por su primer elemento y el resto de la lista. La mitad del *cons* apunta a ese primer elemento; la otra mitad al resto de la lista.

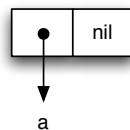


Figura 3.1 Una lista de un elemento, como *cons a nil*.

Cuando aplicamos *cons* a un elemento y una lista vacía, el resultado es un solo *cons* mostrado en la figura 3.1. Como cada *cons* representa un par de apuntes, *car* regresa el objeto apuntado como primer componente del *cons* y *cdr* el segundo:

```

1 CL-USER> (cons 'a nil)
2 (A)
3 CL-USER> (car '(a))
4 A
5 CL-USER> (cdr '(a))
6 NIL

```

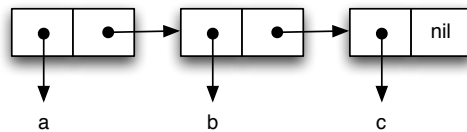


Figura 3.2 Una lista de varios elementos.

Cuando construimos una lista con múltiples componentes, el resultado es una cadena de *conses*. La figura 3.2 ilustra la siguiente construcción:

```

1 CL-USER> (list 'a 'b 'c)
2 (A B C)
3 CL-USER> (cdr (list 'a 'b 'c))
4 (B C)

```

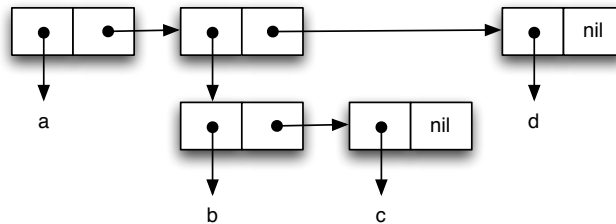


Figura 3.3 Una lista de varios elementos, incluida otra lista.

Las listas pueden tener elementos de cualquier tipo, incluidas otras listas, como lo ilustran las siguientes definiciones y la figura 3.3:

```

1 CL-USER> (list 'a (list 'b 'c) 'd)
2 (A (B C) D)

```

Las listas que no incluyen otras listas, se conocen como **listas planas**. En caso contrario, decimos que se trata de una **lista anidada**.

La función *consp/1* regresa *true* si su argumento es un *cons*, así que podemos definir *listp/1* que regresa *true* si su argumento es una lista como sigue:

```

1 CL-USER> (defun mi-listp (x)
2           (or (null x) (consp x)))
3 MI-LISTP
4 CL-USER> (defun mi-atomp (x)
5           (not (consp x)))
6 MI-ATOMP
7 CL-USER> (mi-listp '(1 2 3))
8 T
9 CL-USER> (mi-atomp '(1 2 3))
10 NIL
11 CL-USER> (mi-listp nil)
12 T
13 CL-USER> (mi-atomp nil)
14 T

```

la definición de *mi-atomp* se basa en que todo lo que no es un *cons* es un **átomo**. Observen que *nil* es lista y átomo a la vez.

3.2. Cons e igualdad

Cada vez que invocamos a *cons*, Lisp reserva memoria para dos apuntadores, así que si llamamos a *cons* con el mismo argumento dos veces, Lisp regresa dos valores que aparentemente son el mismo, pero en realidad se trata de diferentes objetos:

```

1 CL-USER> (eql (cons 1 nil) (cons 1 nil))
2 NIL

```

Sería conveniente contar con una función *mi-eql* que regrese *true* cuando dos listas tienen los mismos elementos, aunque se trate de objetos distintos:

```

1 CL-USER> (defun mi-eql (lst1 lst2)
2           (or (eql lst1 lst2)
3               (and (consp lst1)
4                    (consp lst2)
5                    (mi-eql (car lst1) (car lst2))
6                    (mi-eql (cdr lst1) (cdr lst2))))))
7 MI-EQL
8 CL-USER> (mi-eql (cons 1 nil) (cons 1 nil))
9 T

```

en realidad, lisp cuenta con una función predefinida *equal/2* que cumple con nuestros objetivos. Como la definición de nuestro *eql* lo sugiere, si dos objetos son *eql*, también son *equal*.

Uno de los secretos para comprender Lisp es darse cuenta de que las variables tienen valores, en el mismo sentido en que las listas tienen elementos. Así como los

conses tienen apuntadores a sus elementos, las variables tienen apuntadores a sus valores.

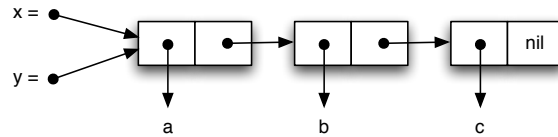


Figura 3.4 Una lista de varios elementos, incluida otra lista.

La diferencia entre Lisp y otros lenguajes de programación donde manipulamos los apuntadores explícitamente, es que en el primero caso, el lenguaje administra los apuntadores por uno. Ya vimos ejemplos de esto relacionados con la creación de listas. Algo similar pasa con las variables. Por ejemplo, si asignamos a dos variables la misma lista:

```

1 CL-USER> (setf x '(a b c))
2 (A B C)
3 CL-USER> (setf y x)
4 (A B C)
5 CL-USER> (eql x y)
6 T

```

¿Qué sucede al hacer la segunda asignación (línea 3)? En realidad, la localidad de memoria asociada a *x* no contiene la lista (*a, b, c*) sino un apuntador a esta lista. El *setf* en cuestión copia ese mismo apuntador a la localidad de memoria asociada a *y*. Es decir, Lisp copia el apuntador relevante, no la lista completa. La figura 3.4 ilustra este caso. Por eso *eql* en la llamada de la línea 5, regresa *true*.

3.3. Construyendo listas

Ya hemos visto ejemplos de construcción de listas con *list* y *cons*. Ahora imaginen que deseamos una función *copia-lista* que una lista como su primer argumento y regresa una copia de ella. La lista resultante tiene los mismos elementos que la lista original, pero está contenida en *conses* diferentes:

```

1 CL-USER> (defun copia-lista (lst)
2           (if (atom lst)
3               lst
4               (cons (car lst) (copia-lista (cdr lst)))))
5
6 COPIA-LISTA
7 CL-USER> (setf x '(a b c)
8           y (copia-lista x))
9 (A B C)
10 CL-USER> (eql x y)

```

```
11 | NIL
```

evidentemente, x y su copia nunca serán *eql* pero si *equal*, al menos que x sea *nil*.

Existe otro constructor de listas que toma como argumento varias listas para construir una sola:

```
1 | CL-USER> (append '(1 2) '(3) '(4))
2 | (1 2 3 4)
```

3.4. Ejemplo: compresión de datos *run-length*

Consideremos un ejemplo para utilizar los conceptos introducidos hasta ahora. RLE (*run-length encoding*) es un algoritmo de compresión de datos muy sencilla. Funciona como los meseros: si los comensales pidieron una tortilla española, otra, otra más y una ensalada verde; el mesero pedirá tres tortillas españolas y una ensalada verde. El código de esta estrategia es como sigue:

```
1 | (defun rle (lst)
2 |   (if (consp lst)
3 |       (compress (car lst) 1 (cdr lst))
4 |       lst))
5 |
6 | (defun compress (elt n lst)
7 |   (if (null lst)
8 |       (list (n-elts elt n))
9 |       (let ((sig (car lst)))
10 |          (if (eql sig elt)
11 |              (compress elt (+ n 1) (cdr lst))
12 |              (cons (n-elts elt n)
13 |                    (compress sig 1 (cdr lst)))))))
14 |
15 | (defun n-elts (elt n)
16 |   (if (> n 1)
17 |       (list n elt)
18 |       elt))
```

Una llamada a *rle* sería como sigue:

```
1 | CL-USER> (rle '(1 1 1 0 1 0 0 0 1))
2 | ((3 1) 0 1 (4 0) 1)
```

Ejercicio sugerido. Programen una función inversa a *rle*, esto es dado una lista que es un código *rle*, esta función regresa la cadena original. Observen que este método de compresión no tiene pérdida de información. Prueben su solución con la ayuda de un generador de listas de n elementos aleatorios.

3.5. Funciones básicas

A continuación definiremos una biblioteca mínima de operaciones sobre listas, que por cuestiones de eficiencia serán declaradas:

```

1 (proclaim '(inline last1 single append1 conc1 mklist))
2 (proclaim '(optimize speed))
3
4 (defun last1 (lst)
5   (car (last lst)))
6
7 (defun single (lst)
8   (and (consp lst) (not (cdr lst))))
9
10 (defun append1 (lst obj)
11   (append lst (list obj)))
12
13 (defun conc1 (lst obj)
14   (nconc lst (list obj)))
15
16 (defun mklist (obj)
17   (if (listp obj) obj (list obj)))

```

La función `last1` regresa el último elemento de una lista. La función predefinida `last` regresa el último *cons* de una lista, no su último elemento. Generalmente obtenemos tal elemento usando `(car (last ...))`. ¿Vale la pena definir una nueva función para una función predefinida? La respuesta es afirmativa cuando la nueva función reemplaza efectivamente a la función predefinida.

Observen que `last1` no lleva a cabo ningún chequeo de error. En general, ninguna de las funciones del curso harán chequeo de errores. En parte esto se debe a que de esta manera los ejemplos serán más claros; y en parte se debe a que no es razonable hacer chequeo de errores en utilidades tan pequeñas. Si intentamos:

```

1 CL-USER> (last1 "prueba")
2 value "prueba" is not of the expected type LIST.
3 [Condition of type TYPE-ERROR]

```

el error es capturado y reportado por la función predefinida `last`. Cuando las utilidades son tan pequeñas, forman una capa de abstracción tan delgada, que comienzan por ser transparentes. Uno puede ver en `last1` para interpretar los errores que ocurren en sus funciones subyacentes.

La función `single` prueba si algo es una lista de un elemento. Los programas Lisp necesitan hacer esta prueba bastantes veces. Al principio, uno está tentado a utilizar la traducción natural del español a Lisp:

```

1 (= (length lst) 1)

```

pero escrita de esta forma, la función sería muy ineficiente.

Las funciones `append1` y `conc1` agregan un elemento al final de una lista, `conc1` de manera destructiva. Estas funciones son pequeñas, pero se usan tantas

veces que vale la pena incluirlas en la librería. De hecho, `append1` ha sido predefinida en muchos dialectos Lisp.

La función `mklist` nos asegura que su argumento sea una lista. Muchas funciones Lisp están escritas para regresar una lista o un elemento. Supongamos que `lookup` es una de estas funciones. Si queremos coleccionar el resultado de aplicar esta función a todos los miembros de una lista, podemos escribir:

```
1 (mapcan #'(lambda (d) (mklist (lookup d)))
2 data)
```

Veamos ahora otros ejemplos de utilidades más grandes que operan sobre listas:

```
1 (defun longer (x y)
2   (labels ((compare (x y)
3             (and (consp x)
4                  (or (null y)
5                      (compare (cdr x) (cdr y))))))
6   (if (and (listp x) (listp y))
7       (compare x y)
8       (> (length x) (length y))))
9
10 (defun filter (fn lst)
11   (let ((acc nil))
12     (dolist (x lst)
13       (let ((val (funcall fn x)))
14         (if val (push val acc))))
15     (nreverse acc)))
16
17 (defun group (source n)
18   (if (zerop n) (error "zero length"))
19   (labels ((rec (source acc)
20             (let ((rest (nthcdr n source)))
21               (if (consp rest)
22                   (rec rest (cons (subseq source 0 n) acc))
23                   (nreverse (cons source acc))))))
24     (if source (rec source nil) nil)))
```

Al comparar la longitud de dos lista, lo más inmediato es usar `(> (length x) (length y))`, pero esto es ineficiente. En particular si una de las listas es mucho más corta que la otra. Lo mejor, es usar `longer` y recorrerlas en paralelo con la función local `compare`, en caso de que `x` e `y` sean listas.

La función `filter` aplica su primer argumento `fn` a cada elemento de la lista `lst` guardando aquellos resultados diferentes a `nil`.

```
1 CL-USER> (filter #'null '(nil t nil t 5 6))
2 (T T)
3 CL-USER> (filter #'(lambda (x)
4                   (when (> x 0) x))
5           '(-1 2 -3 4 -5 6))
6 (2 4 6)
7 > (filter #'(lambda (x)
8           (if (numberp x) (1+ x)))
9         '(a 1 2 b 3 c d 4))
10 (2 3 4 5)
```

Observen el uso del acumulador `acc` en la definición de `filter`. La combinación de `push` y `nreverse` es la forma estándar de acumular una lista en Lisp.

La función `group` agrupa una lista `lst` en sublistas de tamaño `n`:

```
1 | CL-USER> (group '(a b c d e f g) 2)
2 | ((A B) (C D) (E F) (G))
```

Esta función si lleva a cabo un chequeo de error, porque si $n = 0$ `group` entra en un ciclo infinito.

Otras funciones (doblemente recursivas) sobre listas son:

```
1 | (defun flatten (x)
2 |   (labels ((rec (x acc)
3 |             (cond ((null x) acc)
4 |                   ((atom x) (cons x acc))
5 |                   (t (rec (car x) (rec (cdr x) acc))))))
6 |     (rec x nil)))
7 |
8 | (defun prune (test tree)
9 |   (labels ((rec (tree acc)
10 |             (cond ((null tree) (nreverse acc))
11 |                   ((consp (car tree))
12 |                    (rec (cdr tree)
13 |                         (cons (rec (car tree) nil) acc)))
14 |                   (t (rec (cdr tree)
15 |                          (if (funcall test (car tree))
16 |                              acc
17 |                              (cons (car tree) acc))))))
18 |     (rec tree nil)))
```

estas funciones recorren sobre listas anidadas para hacer su trabajo. La primera de ellas, `flatten` regresa la lista de átomos que son elementos de su lista argumento:

```
1 | CL-USER 1 > (flatten '(a (b c) ((d e) f)))
2 | (A B C D E F)
```

La segunda función, `prune` remueve de una lista todo átomo que satisface el predicado `test`, de forma que:

```
1 | CL-USER 2 > (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
2 | (1 (3 (5)) 7 (9))
```

3.6. Mapeos

Otra clase de funciones ampliamente usadas en Lisp son los mapeos, que aplican una función a la secuencia de sus argumentos. La más conocida de estas funciones es `mapcar`. Definiremos otras funciones de mapeo a continuación:

```
1 | (defun map0-n (fn n)
2 |   (mapa-b fn 0 n))
3 |
4 | (defun map1-n (fn n)
```

```

5 (mapa-b fn 1 n)
6
7 (defun mapa-b (fn a b &optional (step 1))
8 (do ((i a (+ i step))
9 (result nil))
10 ((> i b) (nreverse result))
11 (push (funcall fn i) result)))
12
13 (defun map-> (fn start test-fn succ-fn)
14 (do ((i start (funcall succ-fn i))
15 (result nil))
16 ((funcall test-fn i) (nreverse result))
17 (push (funcall fn i) result)))
18
19 (defun mappend (fn &rest lsts)
20 (apply #'append (apply #'mapcar fn lsts)))
21
22 (defun mapcars (fn &rest lsts)
23 (let ((result nil))
24 (dolist (lst lsts)
25 (dolist (obj lst)
26 (push (funcall fn obj) result)))
27 (nreverse result)))
28
29 (defun rmapcar (fn &rest args)
30 (if (some #'atom args)
31 (apply fn args)
32 (apply #'mapcar
33 #'(lambda (&rest args)
34 (apply #'rmapcar fn args))
35 args)))

```

Las primeras tres funciones mapean funciones sobre rangos de números sin tener que hacer `cons` para guardar la lista resultante. Las primeras dos `map0-n` y `map1-n` funcionan con rangos positivos de enteros:

```

1 CL-USER 3 > (map0-n #'1+ 5)
2 (1 2 3 4 5 6)
3
4 CL-USER 4 > (map1-n #'1+ 5)
5 (2 3 4 5 6)

```

Ambas fueron escritas usando `mapa-b` que funciona para cualquier rango de números:

```

1 CL-USER> (mapa-b #'1+ 1 4 0.5)
2 (2 2.5 3.0 3.5 4.0 4.5 5.0)

```

A continuación se implementa un mapeo más general con `map->`, el cual trabaja para cualquier tipo de secuencias de objetos de cualquier tipo. La secuencia comienza con el objeto dado como segundo argumento; el final de la secuencia está dado por el tercer argumento como una función; y los sucesores del primer elemento se generan de acuerdo a la función que se da como cuarto argumento. Con esta función es posible navegar en estructuras de datos arbitrarias, así como operar sobre secuencias de números. Por ejemplo, `mapa-b` puede definirse en términos de `map->` como:

```
1 (defun my-mapa-b (fn a b &optional (step 1))
2   (map-> fn
3     a
4     #'(lambda(x) (> x b))
5     #'(lambda(X) (+ x step))))
```

La función `mapcars` es útil cuando queremos aplicar `mapcar` a varias listas. Las siguientes dos expresiones (`mapcar #'sqrt (append list1 list2)`) y (`mapcars #'sqrt list1 list2`), son equivalentes. Sólo que la segunda versión no hace conses innecesarios.

Finalmente `rmapcar` es un acrónimo para `mapcar` recursivo:

```
1 CL-USER 11 > (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
2 123456789
3 (1 2 (3 4 (5) 6) 7 (8 9))
4
5 CL-USER 12 > (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
6 (11 (22 (33) 44))
```

Capítulo 4

Macros en Lisp

Resumen La definición de una macro es esencialmente el de una función que genera código Lisp –un programa que genera programas. Este mecanismo ofrece grandes posibilidades, pero también da lugar a problemas inesperados. Este capítulo explica como funcionan las macros, ofrece técnicas para escribirlas y probarlas, y revisa el estilo correcto para su definición.

4.1. ¿Cómo funcionan las macros?

Debido a que las macros, al igual que las funciones, pueden invocarse para computar valores de salida, existe una confusión sobre que operadores predefinidos son funciones y cuales macros. La definición de una macro se asemeja a la definición de una función y, de manera informal, incluso los programadores identifican a operadores como `do`, como una función predefinida, cuando en realidad es una macro. Sin embargo, llevar la analogía entre macros y funciones demasiado lejos, puede ocasionar problemas. Las macros tienen un funcionamiento diferente al de las funciones y entender tales diferencias es clave para usar las macros correctamente. Una función produce un resultado, pero una macro produce una expresión que al ser evaluada produce un resultado.

Veamos un ejemplo. Supongan que queremos escribir la macro `nil!` que asigna a su argumento el valor `nil`. Queremos que `(nil! x)` tenga el mismo efecto que `(setq x nil)`. Lo que hacemos es definir `nil!` como una macro que transforma casos de la primera forma en casos de la segunda forma:

```
1 CL-USER> (defmacro nil! (var)
2           (list 'setq var nil))
3 NIL!
4 CL-USER> (nil! x)
5 NIL
6 CL-USER> x
7 NIL
```

Parafraseada al español, la definición anterior le dice a Lisp “Siempre que encuentres una expresión de la forma `(nil! var)`, conviértela en una expresión de la forma `(setq var nil)` y entonces procede a evaluar la expresión resultante”.

La expresión generada por la macro será evaluada en lugar de la llamada original a la macro. Una llamada a una macro es una lista cuyo primer elemento es el nombre de la macro. ¿Qué sucede cuando llamamos a la macro con `(nil! x)` en el toplevel? Lisp identifica a `nil!` como una macro y:

- Construye la expresión especificada por la definición de la macro, y entonces
- Evalúa la expresión en lugar de la llamada original a la macro.

El paso que construye la nueva expresión a ser evaluada se llama *macro-expansión*. Lisp busca en la definición de `nil!` la forma de transformar la llamada original a la macro en una expresión de remplazo. La definición de la macro se aplica a los argumentos de la llamada de manera habitual. En nuestro ejemplo, esto produce la lista `(setq x nil)`.

Tras la macro-expansión, el segundo paso es la *evaluación*. Lisp evalúa la macro-expansión resultante del primer paso, en nuestro ejemplo `(setq x nil)`, y la evalúa como si el programador la hubiese tecleado. Aunque, la evaluación no siempre se produce inmediatamente después de la expansión, como sucede normalmente en el toplevel. Una llamada a una macro que ocurre en la definición de una función será expandida cuando la función sea compilada, pero la expansión – o la expresión que resulta de la expansión, no será evaluada hasta que la función sea llamada.

Muchas de las dificultades propias del uso de las macros pueden evitarse si se tiene clara la diferencia entre macro-expansión y evaluación. Al escribir macros, se debe identificar que computaciones serán ejecutadas en fase de macro-expansión y cuales durante la evaluación. La macro-expansión trabaja con expresiones, y la evaluación lo hace con sus valores.

Algunas veces, la macro-expansión puede ser más complicada que con `nil!`. La expansión de `nil!` era una llamada a una forma especial pre-definida, pero en algunas ocasiones la expansión de una macro puede producir otra macro, como una *matrushka rusa*. En esos casos, la expansión continua hasta que se produce una expresión que no es una llamada a una macro. El proceso puede tener cuantos pasos sean necesarios, a condición de que eventualmente termine.

Muchos lenguajes ofrecen alguna forma de macro, pero el mecanismo ofrecido por Lisp es en particular poderoso. Cuando un archivo de Lisp es compilado, un parser lee el código en él y envía la salida al compilador. He aquí el truco: la salida del parser consiste en una lista de objetos Lisp. Con las macros, podemos manipular el programa mientras está en esta forma intermedia entre el parser y el compilador. Si es necesario, esta manipulación puede ser muy extensa. Una macro generando su expansión tiene a su disposición todo el repertorio de Lisp. De hecho, una macro es realmente una función Lisp que regresa expresiones. La definición de `nil!` contiene únicamente una llamada a `list`, pero es posible que la definición de una macro utilice subprogramas completos para generar la expansión deseada.

La capacidad de cambiar lo que el compilador ve, es casi como tener la capacidad de reescribir el compilador. Podemos agregar cualquier constructor al lenguaje que pueda ser definido mediante transformaciones a constructores existentes.

4.2. Backquote

El *backquote* es una versión especial de nuestro conocido *quote* que puede ser usado para definir moldes de expresiones Lisp. Su uso más común está en la definición de macros. Cuando hacemos apóstrofo invertido sobre una expresión, se comporta igual que *quote*: `'(a b c)` es equivalente a ``(a b c)`. Backquote se vuelve útil cuando se usa conjuntamente con coma “,” y coma-arroba “,@”. Si el apóstrofo invertido es usado para crear un molde, la coma crea las ranuras en el molde. Una lista bajo apóstrofo invertido es equivalente a una llamada a `list` con sus argumentos bajo *quote*. Esto es:

```
1 | CL-USER 3 > (list 'a 'b 'c)
2 | (A B C)
3 | CL-USER 4 > `(a b c)
4 | (A B C)
```

Bajo el alcance de apóstrofo invertido, una coma le dice a Lisp “deten el efecto de *quote*”. Cuando una coma aparece antes de un elemento de una lista, tiene el efecto de cancelar el *quote*, de forma que:

```
1 | CL-USER 5 > (setf a 1 b 2 c 3)
2 | 3
3 | CL-USER 6 > `(a ,(b c))
4 | (A (2 C))
```

El apóstrofo invertido se usa normalmente para construir listas. Cualquier lista generada de esta forma, puede generarse también usando `list` y quotes regulares. La ventaja del apóstrofo invertido es que hace que las expresiones sean más fáciles de leer, debido a que la expresión con apóstrofo invertido es similar a su macro-expansión. Vean las definiciones de `nil!`:

```
1 | (defmacro nil! (var)
2 |   (list 'setq var nil))
3 | (defmacro nil! (var)
4 |   `(setq ,var nil))
```

Aunque en el ejemplo la diferencia es mínima, entre más grande sea la definición de una macro, más relevante es el uso de apóstrofo invertido para hacer su lectura más clara. Veamos un segundo ejemplo más complicado, un `if` numérico donde el primer argumento debe evaluar a un número y los otros tres argumentos son evaluados dependiendo si el número fue positivo, cero o negativo, respectivamente. Por ejemplo:

```

1 CL-USER 8 > (mapcar #'(lambda(x)
2                   (nif x 'p 'c 'n))
3                   '(0 2.5 -8))
4 (C P N)

```

La versión con apóstrofo invertido es:

```

1 (defmacro nif (expr pos zero neg)
2   `(case (truncate (signum ,expr))
3     (1 ,pos)
4     (0 ,zero)
5     (-1 ,neg)))

```

La versión sin apóstrofo invertido es como sigue:

```

1 (defmacro nif (expr pos zero neg)
2   (list 'case
3         (list 'truncate (list 'signum expr))
4         (list 1 pos)
5         (list 0 zero)
6         (list -1 neg)))

```

La coma-arroba es una variante del operador coma. Funciona como la coma normal, sólo que en lugar de insertar el valor de la expresión que antecede, una coma-arroba inserta tal valor removiendo sus paréntesis más externos:

```

1 CL-USER 10 > (setq b '(1 2 3))
2 (1 2 3)
3 CL-USER 11 > `(a ,b c)
4 (A (1 2 3) C)
5 CL-USER 12 > `(a ,@b c)
6 (A 1 2 3 C)

```

Observen que la coma causa que la lista (1 2 3) sea insertada en el lugar de la b, mientras que la coma-arroba, hace que los elementos de la lista 1 2 3 sean insertados en el lugar de b. Existen restricciones adicionales en el uso de coma-arroba:

- Para que sus argumentos sean insertados, la coma-arroba debe ocurrir dentro de una secuencia.
- El objeto a insertar debe ser una lista, o en caso contrario, ser insertado al final de la secuencia destino.

El operador coma-arroba se usa normalmente en macros que toman un número no determinado de argumentos y los pasan a funciones o macros que a su vez reciben un número indeterminado de argumentos. Esta situación es común al definir bloques implícitos. Lisp provee diversos operadores para agrupar código en bloques, incluyendo `block`, `tagbody`, y el más conocido `progn`. Estos operadores rara vez se usan directamente en un programa por lo que se dice que son implícitos –escondidos por las macros.

Un bloque implícito ocurre en cualquier macro predefinida que tenga un cuerpo de varias expresiones. Tanto `let` como `cond` proveen un `progn` explícito. Posiblemente la macro más simple que hace esto es `when`:

```
1 (when (test)
2   (funcion1)
3   (funcion2)
4   obj)
```

si el `test` es verdadero, las expresiones en el cuerpo de la macro son ejecutadas secuencialmente y se regresa el valor de la última expresión `obj`. Como un ejemplo del uso de coma-arroba, definiremos nuestro propio `mi-when`:

```
1 (defmacro mi-when (test &body body)
2   '(if ,test
3     (progn ,@body)))
```

el parámetro `&body` toma un número arbitrario de argumentos y el operador coma-arroba los inserta en un sólo `progn`. La mayoría de las macros de iteración insertan sus argumentos de esta forma.

Aunque las macros normalmente construyen listas, también pueden regresar funciones. Finalmente, el apóstrofo invertido puede usarse en cualquier expresión Lisp, no sólo en las macros:

```
1 (defun hola (nombre)
2   `(hola ,nombre))
```

4.3. Definiendo macros simples

Comencemos por escribir una macro que sea una variante de la función predefinida `member`. Por default, ésta función utiliza `eq` para probar igualdad. Si se quiere probar membresía usando `eq`, esto debe indicarse explícitamente:

```
1 (member obj lst :test #'eq)
```

Si hacemos esto muchas veces, nos gustaría tener una variante de `member` que siempre use `eq`. Aunque normalmente definiríamos esta versión como una función *inline* (ver clase anterior), su definición es un buen ejercicio sobre codificación de macros.

El método para definir una macro es como sigue: se comienza con la llamada a la macro que queremos definir. Escribanla en un papel y abajo escriban la expresión que quieren producir con la macro:

```
1 llamada: (mem-eq obj lst)
2 expansión: (member obj lst :test #'eq)
```

La llamada nos sirve para definir los parámetros de la macro. En este caso, como necesitamos dos argumentos, el inicio de la macro será:

```
1 (defmacro mem-eq (obj lst)
```

Ahora volvamos a las dos expresiones iniciales. Para cada argumento en la llamada a la macro, tracen un línea hacia donde son insertadas en la expansión. Para escribir el cuerpo de la macro presten atención a estas líneas paralelas. Comiencen el cuerpo con un apóstrofo invertido. Ahora lean la expansión expresión por expresión. Donde quiera que encuentre un paréntesis que no es parte de los argumentos de llamada de la macro, pongan un paréntesis en la definición de la macro. Así tenemos que para cada expresión:

1. Si no hay una línea conectado la expresión en la llamada, entonces escribir la expresión tal cual en el cuerpo de la macro.
2. Si hay una conexión, escriban la expresión precedida por una coma.

```
1 (defmacro mem-eq (obj lst)
2   `(member ,obj ,lst :test #'eq))
```

Hasta ahora hemos escrito macros que tienen un número determinado de argumentos. Ahora supongan que queremos escribir la macro `while` que toma una expresión de prueba `test` y algún cuerpo que ejecutará repetidamente mientras el `test` sea verdadero. Esta macro requiere modificar el método anterior ligeramente. Comencemos por escribir la llamada a la macro:

```
1 (defmacro while (test &body body)
```

Ahora escriban la expansión deseada y como el caso anterior, sin embargo cuando tengan una secuencia de argumentos bajo `rest` o `body`, tratenlos como un grupo, dibujando una sola línea para toda la secuencia. Y claro, al insertar la expresión en la definición de la macro, hay que recurrir a coma-arroba:

```
1 (defmacro while (test &body body)
2   `(do ()
3     ((not ,test)
4      ,@body))
```

4.4. Probando la expansión de las macros

Una vez que hemos escrito una macro ¿Cómo podemos probarla? Las macros simples, como `mem-eq` pueden probarse directamente, observando si su salida es la esperada. Para macros más complejas, es necesario poder observar si la expansión ha sido correcta. Para ello `lisp` provee las funciones predefinidas `macroexpand` y

`macroexpand-1`. La primera muestra como la macro se expandiría antes de ser evaluada. Si la macro hace uso de otras macros, esta revisión de la expansión es de poca utilidad. La función `macroexpand-1` muestra la expansión de un sólo paso. Veamos un ejemplo basado en `while`:

```

1 CL-USER 2 > (pprint (macroexpand '(while (puedas) (rie))))
2 (BLOCK NIL
3   (LET ()
4     (DECLARE (IGNORABLE))
5     (DECLARE)
6     (TAGBODY
7       #:G747 (WHEN (NOT (PUEDAS)) (RETURN-FROM NIL NIL))
8       (RIE)
9       (GO #:G747))))
10
11 CL-USER 3 > (pprint (macroexpand-1 '(while (puedas) (rie))))
12 (DO () ((NOT (PUEDAS))) (RIE))

```

Observen que la expansión completa es más difícil de leer, mientras que la producida por `macroexpand-1` es más útil en este caso. La expansión depende de la implementación de Lisp que estén usando, en este caso LispWorks 5.1.2. Otros Lisp pueden implementar `do` en términos de `unless` en lugar de `unless`. Si vamos a hacer esto muchas veces, nos conviene definir una macro:

```

1 (defmacro mac (expr)
2   `(pprint (macroexpand-1 ',expr)))

```

de forma que podemos evaluar ahora:

```

1 CL-USER> (mac (while (puedas) rie))
2 (DO () ((NOT (PUEDAS))) RIE)
3 ; No value

```

La expansión obtenida puede reevaluarse en el TOP-LEVEL para experimentar con la macro:

```

1 CL-USER> (setq aux (macroexpand-1 '(mem-eq 'a '(a b c))))
2 (MEMBER 'A '(A B C) :TEST #'EQ)
3 CL-USER> (eval aux)
4 (A B C)

```

Estas herramientas no sólo son útiles para probar las macros, sino para aprender a escribir macros correctamente. Como he mencionado, numerosas facilidades de Lisp están implementadas como macros. Es posible entonces usar `macroexpand-1` para ver que forma intermedia generan esas expresiones!

```

1 CL-USER> (mac (when t nil))
2 (IF T (PROGN NIL))
3 ; No value

```

4.5. Ejemplos

Una pequeña librería de macros:

```

1 (defmacro for (var start stop &body body)
2   (let ((gstop (gensym)))
3     `(do ((,var ,start (1+ ,var))
4           (,gstop ,stop))
5         (> ,var ,gstop)
6         ,@body)))
7
8 (defmacro in (obj &rest choices)
9   (let ((insym (gensym)))
10    `(let ((,insym ,obj))
11        (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
12                    choices))))))
13
14 (defmacro random-choice (&rest exprs)
15   `(case (random ,(length exprs))
16     ,@(let ((key -1))
17         (mapcar #'(lambda (expr)
18                   `',(incf key) ,expr)
19               exprs))))
20
21 (defmacro avg (&rest args)
22   `(/ (+ ,@args) ,(length args)))
23
24 (defmacro with-gensym (syms &body body)
25   `(let , (mapcar #'(lambda (s)
26                     `',(s (gensym)))
27                 syms)
28       ,@body))
29
30 (defmacro aif (test then &optional else)
31   `(let ((it ,test))
32       (if it ,then ,else)))

```

y sus corridas:

```

1 CL-USER 3 > (for x 1 8 (princ x))
2 12345678
3 NIL
4 CL-USER 4 > (in 3 1 2 3)
5 T
6 CL-USER 5 > (random-choice 1 2 3)
7 1
8 CL-USER 6 > (random-choice 1 2 3)
9 3
10 CL-USER 7 > (random-choice 1 2 3)
11 3
12 CL-USER 8 > (random-choice 1 2 3)
13 1
14 CL-USER 9 > (random-choice 1 2 3)
15 1
16 CL-USER 10 > (random-choice 1 2 3)
17 2
18 CL-USER 11 > (avg 2 4 8)
19 14/3

```

Capítulo 5

Una aplicación sencilla: Significancia de los codones en el DNA

Resumen En esta sesión revisaremos algunos aspectos no funcionales de Lisp, como las operaciones de E/S y el diseño de una interfaz gráfica en el contexto de una aplicación para la bioinformática. La idea es cuantificar la significancia de los codones del código genético y sus propiedades físico-químicas [8]. La identidad de los aminoácidos expresados por los codones (tripletes de nucleótidos) en el código genético, parece depender de la posición de los nucleótidos en el codón, así como de sus propiedades físico-químicas. La literatura propone diferentes ordenes de relevancia tanto para los nucleotidos, como para sus propiedades. El artículo sigue la estrategia de utilizar matrices de similitud para cuantificar la relevancia. La metodología resumida es la siguiente: cada nucleotido tiene varios mapeos posibles que se aplican sistemáticamente al código genético estándar para cuantificar que posición resulta más afectada por estas mutaciones. Por más afectado queremos decir que las matrices de similitud nos dicen que tan parecido es el nuevo aminoácido al original. Se computa la significancia promedio en estos términos y se comparan los resultados para establecer la relevancia buscada.

5.1. Paquetes

Siendo una aplicación más grande que las tareas del curso, conviene empaquetar esta aplicación. Los paquetes nos permiten organizar nuestros programas para evitar conflictos en los símbolos usados tanto en el REPL como por otras librerías. Además nos permiten configurar lo que sería el API público de una librería.

Antes de usar los paquetes es necesario entender sus limitaciones. Primero, los paquetes no proveen control directo sobre quién llama a qué función o accesa tal variable. El paquete sólo provee un control básico sobre espacios de nombres al contralrar como REPL convierte los nombres textuales en símbolos, pero más que el *reader* es el *evaluator* quien lleva a cabo esta tarea. Por eso no tiene sentido hablar de exportar una función o una variable desde un paquete. Podemos exportar

símbolos para hacer ciertos nombres más fáciles de referenciar, pero el sistema de paquetes no permite restringir la manera en que esos nombres son usados.

Con esto en mente, podemos comenzar a experimentar. Los paquetes en Lisp se definen con la macro `defpackage` que permite no sólo crear un paquete nuevo, sino especificar que paquetes serán usados por el paquete nuevo, qué símbolos exporta, qué símbolos importa de otros paquetes y crear símbolos *shadow* para la resolución de conflictos. Si queremos definir el paquete para nuestra aplicación bajo el nombre de `QInfo` y especificar que usaremos el `CAPI` de `LispWorks`, entonces comenzaremos nuestro programa por:

```

1 (defpackage "QINFO"
2   (:add-use-defaults t)
3   (:use "CAPI"))
4
5 (in-package "QINFO")

```

La `s`-expresión `(in-package 'QINFO)` hace que el paquete por default sea éste, en lugar de `CL-USER`. Si ustedes evalúan esta `s`-expresión en el `REPL`, verán que el prompt cambia a `QINFO>` para indicar que los símbolos definidos en el paquete están accesibles al usuario en el `REPL`.

La variable especial `*package*` tiene el valor del paquete actual, por ejemplo:

```

1 CL-USER 6 > *package*
2 #<The COMMON-LISP-USER package, 2/16 internal, 0/4 external>
3 CL-USER 7 > (in-package "QINFO")
4 #<The QINFO package, 101/256 internal, 0/16 external>
5 QINFO 8 > *package*
6 #<The QINFO package, 101/256 internal, 0/16 external>

```

Los mapeos de nombres a símbolos dentro de un paquete pueden ser de dos clases: externos e internos. Dentro del paquete, un nombre se refiere a un símbolo o no, si es que hace referencia a un símbolo, el nombre será externo o interno, pero no ambos. Los símbolos externos son parte de la interfaz pública del paquete. Los símbolos se vuelven externos si son exportados por el paquete. Un símbolo tiene el mismo nombre, sin importar el paquete actual, sólo que en algunas ocasiones será un símbolo interno y en otras uno externo.

Los paquetes pueden organizarse en capas, usando `use-package` para heredar mapeos de otros paquetes, en nuestro ejemplo es el caso de `(:use CAPI)`. Para más información sobre paquetes, vean el capítulo 21 del libro de P. Seibel [22].

5.2. La resolución del problema

Siguiendo a Guerra et.al. [8], sabemos que necesitamos representar el código genético estándar y las matrices de similitud en nuestro programa. Como éstas serán utilizadas en todo el programa, podemos definir las como variables globales. Por ejemplo, podemos definir la matriz de similitud de Dayhoff como una lista de listas:

```

1 (defvar *pam250*
2   ;; Dayhoff PAM250 (percent accepted mutations) as reported
3   ;; by Mac Donaill, Molecular Simulation 30(5) p.269
4   '( ( 2 -2 0 0 -2 0 0 1 -1 -1 -2 -1 -1 -4 1 1 1 -6 -3 0)
5     (-2 6 0 -1 -4 1 -1 -3 2 -2 -3 3 0 -4 0 0 -1 2 -4 -2)
6     ( 0 0 2 2 -4 1 1 0 2 -2 -3 1 -2 -4 -1 1 0 -4 -2 -2)
7     ( 0 -1 2 4 -5 2 3 1 1 -2 -4 0 -3 -6 -1 0 0 -7 -4 -2)
8     (-2 -4 -4 -5 12 -5 -5 -3 -3 -2 -6 -5 -5 -4 -3 0 -2 -8 0 -2)
9     ( 0 1 1 2 -5 4 2 -1 3 -2 -2 1 -1 -5 0 -1 -1 -5 -4 -2)
10    ( 0 -1 1 3 -5 2 4 0 1 -2 -3 0 -2 -5 -1 0 0 -7 -4 -2)
11    ( 1 -3 0 1 -3 -1 0 5 -2 -3 -4 -2 -3 -5 -1 1 0 -7 -5 -1)
12    (-1 2 2 1 -3 3 1 -2 6 -2 -2 0 -2 -2 0 -1 -1 -3 0 -2)
13    (-1 -2 -2 -2 -2 -2 -2 -3 -2 5 2 -2 2 1 -2 -1 0 -5 -1 4)
14    (-2 -3 -3 -4 -6 -2 -3 -4 -2 2 6 -3 4 2 -3 -3 -2 -2 -1 2)
15    (-1 3 1 0 -5 1 0 -2 0 -2 -3 5 0 -5 -1 0 0 -3 -4 -2)
16    (-1 0 -2 -3 -5 -1 -2 -3 -2 2 4 0 6 0 -2 -2 -1 -4 -2 2)
17    (-3 -4 -3 -6 -4 -5 -5 -5 -2 1 2 -5 0 9 -5 -3 -3 0 7 -1)
18    ( 1 0 0 -1 -3 0 -1 0 0 -2 -3 -1 -2 -5 6 1 0 -6 -5 -1)
19    ( 1 0 1 0 0 -1 0 1 -1 -1 -3 0 -2 -3 1 2 1 -2 -3 -1)
20    ( 1 -1 0 0 -2 -1 0 0 -1 0 -2 0 -1 -3 0 1 3 -5 -3 0)
21    (-6 2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4 0 -6 -2 -5 17 0 -6)
22    (-3 -4 -2 -4 0 -4 -4 -5 0 -1 -1 -4 -2 7 -5 -3 -3 0 10 -2)
23    ( 0 -2 -2 -2 -2 -2 -2 -1 -2 4 2 -2 2 -1 -1 -1 0 -6 -2 4)
24  ) "PAM250 matrix")

```

O bien la representación del código genético universal, puede ser una lista de listas, donde las listas miembro son de la forma triplete (otra lista) y aminoácido:

```

1 (defvar *gc-tensor*
2   ;; The genetic code tensor A (universal genetic code)
3   '(((t t t) phe) ((t c t) ser) ((t a t) tyr) ((t g t) cys)
4     ((t t c) phe) ((t c c) ser) ((t a c) tyr) ((t g c) cys)
5     ((t t a) leu) ((t c a) ser) ((t a a) ter) ((t g a) ter)
6     ((t t g) leu) ((t c g) ser) ((t a g) ter) ((t g g) trp)
7     ((c t t) leu) ((c c t) pro) ((c a t) his) ((c g t) arg)
8     ((c t c) leu) ((c c c) pro) ((c a c) his) ((c g c) arg)
9     ((c t a) leu) ((c c a) pro) ((c a a) gln) ((c g a) arg)
10    ((c t g) leu) ((c c g) pro) ((c a g) gln) ((c g g) arg)
11    ((a t t) ile) ((a c t) thr) ((a a t) asn) ((a g t) ser)
12    ((a t c) ile) ((a c c) thr) ((a a c) asn) ((a g c) ser)
13    ((a t a) ile) ((a c a) thr) ((a a a) lys) ((a g a) arg)
14    ((a t g) met) ((a c g) thr) ((a a g) lys) ((a g g) arg)
15    ((g t t) val) ((g c t) ala) ((g a t) asp) ((g g t) gly)
16    ((g t c) val) ((g c c) ala) ((g a c) asp) ((g g c) gly)
17    ((g t a) val) ((g c a) ala) ((g a a) glu) ((g g a) gly)
18    ((g t g) val) ((g c g) ala) ((g a g) glu) ((g g g) gly))
19  "universal genetic code - tensor A")

```

La matriz de similitud de Dayhoff *pam250* nos dice la similitud entre el aminoácido del renglón y la columna. Es necesario pues una función de indexación para buscar los aminoácidos en esta representación. Como los aminoácidos pueden ser referidos por una letra o tres letras, la función queda escrita como sigue:

```

1 (defun index (aa)
2   ;; Get the index in a matrix
3   (cond ((or (equal aa 'A)
4             (equal aa 'ALA)) 0)
5         ((or (equal aa 'R)
6             (equal aa 'ARG)) 1)
7         ((or (equal aa 'N)
8             (equal aa 'ASN)) 2)

```



```

11         ((equal matrix *blosum62*) 0)
12         ((equal matrix *robersy-grau*)
13          (nth (index aaj)
14              (nth (index aai) matrix)))
15         (t (error "matrix not defined")))
16     ((and (equal aai 'ter) ;; ter to aminoacid transition
17          (not (equal aaj 'ter))) (cond ((equal matrix *pam250*) -8)
18                                       ((equal matrix *codegen*) 6)
19                                       ((equal matrix *miyazawa*) -1.01)
20                                       ((equal matrix *prlic*) 0)
21                                       ((equal matrix *blosum62*) 0)
22                                       ((equal matrix *robersy-grau*)
23                                        (nth (index aaj)
24                                            (nth (index aai) matrix)))
25                                       (t (error "matrix not defined")))
26     )
27     ((and (not (equal aai 'ter)) ;; aminoacid to ter transition
28          (equal aaj 'ter)) (cond ((equal matrix *pam250*) -8)
29                                  ((equal matrix *codegen*) 6)
30                                  ((equal matrix *miyazawa*) -1.01)
31                                  ((equal matrix *prlic*) 0)
32                                  ((equal matrix *blosum62*) 0)
33                                  ((equal matrix *robersy-grau*)
34                                   (nth (index aaj)
35                                       (nth (index aai) matrix)))
36                                  (t (error "matrix not defined")))
37     )
38     (t (nth (index aaj)
39            (nth (index aai) matrix))))))

```

De esta forma podemos saber que tan significativo es una mutación de `val` a `arg` según la matriz `*pam250*` (default) o la matriz `*blosum62*`, u otros:

```

1 QInfo 2 > (get-dist 'val 'arg)
2 -2
3 QInfo 3 > (get-dist 'val 'arg *blosum62*)
4 -3

```

Ahora necesitamos funciones para manejar los nucleótidos y los aminoácidos. Esto es, dado un aminoácido, obtener sus tres nucleótidos y viceversa:

```

1 (defun aa-to-nnn (aa)
2   ;; Gets the nucleotides of the aminoacid
3   (remove-if #'(lambda (aminoacid) (not (equal (cadr aminoacid) aa)))
4             *gc-tensor*))
5
6 (defun nnn-to-aa (nnn)
7   ;; Gets the aminoacid from the nucleotides
8   (car (member-if #'(lambda (aminoacid) (equal (car aminoacid) nnn))
9                 *gc-tensor*)))

```

observen que estas funciones operan sobre la tabla `*gc-tensor*` por lo que explota su estructura para resolver la búsqueda. Por ejemplo:

```

1 QInfo 8 > (aa-to-nnn 'val)
2 (((G T T) VAL) ((G T C) VAL) ((G T A) VAL) ((G T G) VAL))
3 QInfo 9 > (nnn-to-aa '(g t t))
4 ((G T T) VAL)

```

Problema 1 *¿Puede mejorarse la definición de `get-dist` para hacerla más legible?*

Problema 2 *La funciones propuestas está diseñada para trabajar con la representación propuesta para las matrices de similitud. ¿Pueden pensar en otra representación para las matrices? ¿Cómo sería entonces `get-dist`?*

El resto de las funciones están autodocumentadas en el código de esta sesión. Lo que resta es crear funciones para comunicar los resultados obtenidos: la interfaz con el usuario. Podemos hacer uso de `format`, por ejemplo, la siguiente función imprime un tensor:

```

1 (defun print-tensor (tensor)
2   (cond ((null tensor) t)
3         (t (progn
4             (format t "~a ~a ~a ~a ~%"
5                   (car tensor) (cadr tensor)
6                   (caddr tensor) (caddr tensor))
7             (print-tensor (cddddr tensor)))
8         )))

```

de forma que:

```

1 QINFO 42 > (print-tensor *gc-tensor*)
2 ((T T T) PHE) ((T C T) SER) ((T A T) TYR) ((T G T) CYS)
3 ((T T C) PHE) ((T C C) SER) ((T A C) TYR) ((T G C) CYS)
4 ((T T A) LEU) ((T C A) SER) ((T A A) TER) ((T G A) TER)
5 ((T T G) LEU) ((T C G) SER) ((T A G) TER) ((T G G) TRP)
6 ((C T T) LEU) ((C C T) PRO) ((C A T) HIS) ((C G T) ARG)
7 ((C T C) LEU) ((C C C) PRO) ((C A C) HIS) ((C G C) ARG)
8 ((C T A) LEU) ((C C A) PRO) ((C A A) GLN) ((C G A) ARG)
9 ((C T G) LEU) ((C C G) PRO) ((C A G) GLN) ((C G G) ARG)
10 ((A T T) ILE) ((A C T) THR) ((A A T) ASN) ((A G T) SER)
11 ((A T C) ILE) ((A C C) THR) ((A A C) ASN) ((A G C) SER)
12 ((A T A) ILE) ((A C A) THR) ((A A A) LYS) ((A G A) ARG)
13 ((A T G) MET) ((A C G) THR) ((A A G) LYS) ((A G G) ARG)
14 ((G T T) VAL) ((G C T) ALA) ((G A T) ASP) ((G G T) GLY)
15 ((G T C) VAL) ((G C C) ALA) ((G A C) ASP) ((G G C) GLY)
16 ((G T A) VAL) ((G C A) ALA) ((G A A) GLU) ((G G A) GLY)
17 ((G T G) VAL) ((G C G) ALA) ((G A G) GLU) ((G G G) GLY)
18 T

```

pero lo que realmente necesitamos es una interfaz gráfica que nos permita seleccionar la matriz de similitud que deseamos usar y el cómputo que queremos llevar a cabo.

5.3. Una interfaz gráfica

Después de trabajar con nuestro sistema QInfo en el REPL, lo ideal sería programar una interfaz gráfica (GUI) para esta aplicación. Aunque el diseño puede ser variado, la GUI debería permitirnos controlar la matriz de similitud a usar y el tipo de cómputo a llevar a cabo. Supongan que la interfaz gráfica deseada es como

se muestra en la figura 5.1. ¿Cómo podemos implementar esta GUI? Eso depende de la implementación de Lisp utilizada y la librería de gráficos elegida. En lo que sigue utilizaremos LispWorks 5.1 y su CAPI (*Common Application Programmer's Interface*).

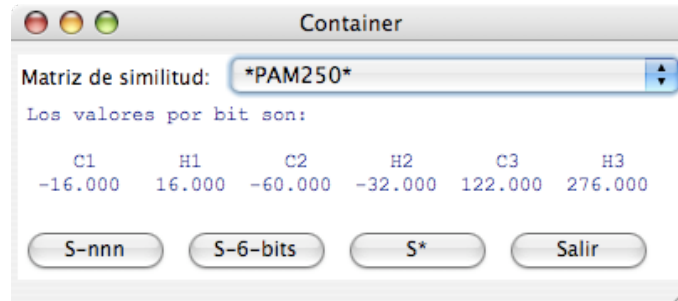


Figura 5.1 La interfaz gráfica de QInfo

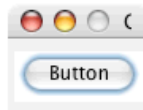
El CAPI es una librería para implementar interfaces de aplicaciones basadas en ventanas, portables a diferentes sistemas operativos. La interfaz se modela usando CLOS y para ello el CAPI provee cuatro clases de objetos básicos que incluyen interfaces, menus, paneles y formatos (*layouts*). La ayuda de Lispworks provee una descripción detallada de estas clases de objetos.

Para comenzar, necesitamos tener acceso a los símbolos de la librería CAPI, por eso al declarar el paquete QINFO, le pedimos que haga uso de esta librería con la línea (`:use CAPI''`).

Una primera aproximación para crear la ventana principal y sus objetos es usando `contain`. En realidad lo ideal es definir interfaces con `define-interface` y desplegarlas con `display`, pero para el desarrollo rápido y pruebas puede usarse `contain`. Para crear una ventana con un botón se puede escribir:

```
1 (make-instance 'push-button
2           :data "Button")
3
4 (contain *)
```

y LispWorks desplegará la ventana:



`contain` provee un formato por default para cada elemento del CAPI que se especifique. En este caso se crea un botón y éste es desplegado dentro de la interfaz.

Para que el botón tenga funcionalidad es necesario especificar el código que ejecutará mediante el slot `:callback`. Si queremos que nuestro botón despliegue en un panel el mensaje `Hola todos`, podemos escribir:

```

1 (make-instance 'push-button
2   :data "Hello"
3   :callback
4     #'(lambda (&rest args)
5         (display-message
6           "Hello World")))
7 (contain *)

```

De esta forma podemos definir los botones de nuestra interfaz:

```

1 (setq run-s-nnn (make-instance 'push-button
2   :text "S-nnn"
3   :callback 's-nnn-message
4   :visible-min-width '(:character 7)))
5
6 (setq run-s-6-bits (make-instance 'push-button
7   :text "S-6-bits"
8   :callback 's-6-bits-message
9   :visible-min-width '(:character 7)))
10
11 (setq run-s* (make-instance 'push-button
12   :text "S*"
13   :callback 's*-message
14   :visible-min-width '(:character 7)))
15
16 (setq exit (make-instance 'push-button
17   :text "Salir"
18   :callback #'(lambda (data interface)
19                 (quit-interface interface))
20   :visible-min-width '(:character 7)))

```

donde las funciones que ejecutal los callback están definidas por:

```

1 (defun s-nnn-message (data interface)
2   (declare (ignore data interface))
3   (apply-in-pane-process
4     output
5     #'(setf display-pane-text)
6     (format nil "Los valores por codón son: ~%~%~9:@<S1~>~9
7             :@<S2~>~9:@<S3~>~%~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>~9"
8             (S-nnn 1 (eval *option*))
9             (S-nnn 2 (eval *option*))
10            (S-nnn 3 (eval *option*)))
11     output))
12
13 (defun s-6-bits-message (data interface)
14   (declare (ignore data interface))
15   (apply-in-pane-process
16     output
17     #'(setf display-pane-text)
18     (format nil "Los valores por bit son:~%~%~9:@<C1~>~9:@<H1~>~9:
19             @<C2~>~9:@<H2~>~9:@<C3~>~9:@<H3~>~%~9:@<~6,3F~>~9:
20             @<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>~9:
21             @<~6,3F~>~9"
22             (S-6bits 1 (eval *option*))
23             (S-6bits 2 (eval *option*))
24             (S-6bits 3 (eval *option*))
25             (S-6bits 4 (eval *option*)))

```

```

26         (S-6bits 5 (eval *option*))
27         (S-6bits 6 (eval *option*)))
28     output))
29
30 (defun s*-message (data interface)
31 (declare (ignore data interface))
32 (apply-in-pane-process
33  output
34  #'(setf display-pane-text)
35  (format nil "Los valores por codón son: ~%~%~9:@<S1~>~9:@<S2~>
36          ~9:@<S3~>~%~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>")
37          (S* 1 (eval *option*))
38          (S* 2 (eval *option*))
39          (S* 3 (eval *option*)))
40  output))

```

Ahora podemos formatear los botones en un renglón:

```

1 (setq buttons
2   (make-instance 'row-layout
3     :description (list run-s-nnn run-s-6-bits run-s* exit)))

```

Para elegir la matriz de similitud que se usará, podemos usar un panel de opciones:

```

1 (defun set-option (data interface)
2   (setq *option* data))
3
4 (setq options (make-instance 'option-pane
5   :items *options*
6   :selected-item *pam250*
7   :selection-callback 'set-option
8   :title "Matriz de similitud: "))

```

y para desplegar los resultados un panel de display llamado `output` porque es ahí donde las acciones de los botones despliegan sus resultados:

```

1 (setq output
2   (make-instance 'display-pane
3     :font (gp:make-font-description
4       :family "Courier New"
5       :size 12)
6     :foreground :navy
7     :text '("Bienvenido a QInfo UV/DIA guerra")
8     :visible-min-height '(:character 5)))

```

Finalmente `contain` despliega toda la interfaz:

```

1 (contain
2   (make-instance 'column-layout
3     :description (list options output buttons)))

```

5.4. Creando un ejecutable

Una vez que hemos programado la interfaz gráfica de nuestra aplicación, y si contamos con la versión profesional de LispWorks, podemos crear un ejecutable de nuestra aplicación. Para ello es necesario escribir un script dependiente del sistema operativo que estamos usando. En el caso de Mac OS X, el script es como sigue:

```
1   ;;; Automatically generated delivery script
2
3   (in-package "CL-USER")
4
5   (load-all-patches)
6
7   ;;; Load the application:
8
9   (compile-file "proyecto.lsp")
10  (load "proyecto")
11
12  ;;; Load the example file that defines WRITE-MACOS-APPLICATION-BUNDLE
13  ;;; to create the bundle.
14
15  (compile-file (sys:example-file
16               "configuration/macos-application-bundle.lisp") :load t)
17
18  (deliver '|QInfo|::start
19         (when (save-argument-real-p)
20             (write-macos-application-bundle
21              "bioinfo05.app")))
22  0
23  :interface :capi)
```

Al llamar a este script desde LispWorks, creamos una aplicación ejecutable que incluye la interfaz gráfica.

Capítulo 6

Una aplicación más elaborada: ID3 en Lisp

Resumen Este capítulo introduce el material base para el proyecto final del curso: implementaciones concurrentes y distribuidas de la inducción de bosques de decisión. Los bosques de decisión son conjuntos de árboles de decisión. Continuaremos nuestras prácticas de Lisp con la implementación de un algoritmo de aprendizaje automático bien conocido: ID3 [19, 20]. Si bien este método de inducción de árboles de decisión ha sido abordado en sus cursos de Aprendizaje Automático y Metodologías de Programación I, haremos una revisión breve de él para entrar en materia. Este capítulo ilustrará también algunos aspectos no funcionales de Lisp introducidos en el capítulo anterior, como la lectura de archivos, el uso de librerías y la programación de interfaces gráficas.

6.1. Árboles de decisión

Un árbol de decisión representa una hipótesis como una conjunción de disyunciones sobre atributos proposicionales (Aunque puede ser extendido a representaciones de primer orden [1]). Cada rama del árbol representa una conjunción de pares atributo-valor y el árbol completo es la disyunción de esas conjunciones.

La figura 6.1 muestra un árbol de decisión típico. Cada nodo del árbol está conformado por un *atributo* y puede verse como la pregunta: ¿Qué valor tiene este atributo en el caso que vamos a clasificar? Las ramas que salen de cada nodo corresponden a los posibles valores del atributo en cuestión. Los nodos que no tienen hijos, se conocen como *hojas* y representan un valor de la *clase* que se quiere predecir.

Los árboles de decisión pueden representarse naturalmente en Lisp como una lista de listas:

```
CL-USER> (setq arbol '(cielo
                       (soleado (humedad
                                  (normal si)
                                  (alta no)))
                       (nublado si)
                       (lluvia (viento
```

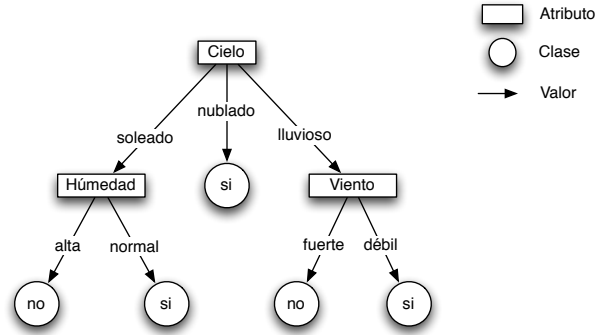


Figura 6.1 Un ejemplo de árbol de decisión para el concepto “buen día para jugar tenis”.

```

(fuerte no)
(debil si))))))
(CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI) (LLUVIA (VIENTO (
FUERTE NO) (DEBIL SI))))))
CL-USER> (root arbol)
CIELO
CL-USER> (sub-tree arbol 'soleado)
(HUMEDAD (NORMAL SI) (ALTA NO))
CL-USER> (sub-tree arbol 'nublado)
SI
  
```

Podemos escribir una función para desplegar el árbol de manera más legible:

```

1 (defun print-tree (tree &optional (depth 0))
2   (mytab depth)
3   (format t "~A%" (first tree))
4   (loop for subtree in (cdr tree) do
5     (mytab (+ depth 1))
6     (format t "- `A" (first subtree))
7     (if (atom (second subtree))
8       (format t " -> `A%" (second subtree))
9       (progn (terpri)(print-tree (second subtree) (+ depth 5))))))
10
11 (defun mytab (n)
12   (loop for i from 1 to n do (format t " ")))
  
```

De forma que:

```

CL-USER> (print-tree arbol)
CIELO
- SOLEADO
  HUMEDAD
  - NORMAL -> SI
  - ALTA -> NO
- NUBLADO -> SI
- LLUVIA
  VIENTO
  - FUERTE -> NO
  - DEBIL -> SI
NIL
  
```

Una vez que hemos adoptado esta representación, es posible definir funciones de acceso y predicados adecuados para la manipulación del árbol. Por ejemplo:

```

1 (defun root(tree)
2   (car tree))
3
4 (defun sub-tree(tree attribute-value)
5   (second (assoc attribute-value (cdr tree))))
6
7 (defun leaf(tree)
8   (atom tree))

```

Tal que:

```

CL-USER> (root arbol)
CIELO
CL-USER> (sub-tree arbol 'soleado)
(HUMEDAD (NORMAL SI) (ALTA NO))
CL-USER> (sub-tree arbol 'nublado)
SI
CL-USER> (leaf (sub-tree arbol 'soleado))
NIL
CL-USER> (leaf (sub-tree arbol 'nublado))
T

```

La función predefinida `assoc` busca el subárbol asociado al valor del atributo en cuestión, cuyo segundo elemento es el subárbol propiamente dicho. Consideren el siguiente ejemplo de su uso:

```

CL-USER> (assoc 'uno '((uno 1) (dos 2) (tres 3)))
(UNO 1)
CL-USER> (assoc 'dos '((uno 1) (dos 2) (tres 3)))
(DOS 2)

```

6.2. Representación de los ejemplos de entrenamiento

La idea de ID3 es inducir árboles de decisión a partir de un conjunto de ejemplos de entrenamiento, casos cuyo valor de clase conocemos, de forma que el árbol explique los ejemplos vistos y pueda predecir casos cuyo valor de clase desconocemos. Para el árbol mostrado en la sección anterior los ejemplos con que fue inducido se muestran en la tabla 6.1.

Basados en la representación elegida para los árboles de decisión, podríamos representar los ejemplos como una lista de listas de valores para sus atributos:

```

CL-USER> (setq
ejemplos
'((SOLEADO CALOR ALTA DEBIL NO)
(SOLEADO CALOR ALTA FUERTE NO)
(NUBLADO CALOR ALTA DEBIL SI)
(LLUVIA TEMPLADO ALTA DEBIL SI)
(LLUVIA FRIO NORMAL DEBIL SI))

```

Día	Cielo	Temperatura	Humedad	Viento	Jugar-tenis?
1	soleado	calor	alta	débil	no
2	soleado	calor	alta	fuerte	no
3	nublado	calor	alta	débil	si
4	lluvia	templado	alta	débil	si
5	lluvia	frío	normal	débil	si
6	lluvia	frío	normal	fuerte	no
7	nublado	frío	normal	fuerte	si
8	soleado	templado	alta	débil	no
9	soleado	frío	normal	débil	si
10	lluvia	templado	normal	débil	si
11	soleado	templado	normal	fuerte	si
12	nublado	templado	alta	fuerte	si
13	nublado	calor	normal	débil	si
14	lluvia	templado	alta	fuerte	no

Cuadro 6.1 Conjunto de ejemplos de entrenamiento para el concepto objetivo jugar-tenis? en ID3, por Tom M. Mitchel [17].

```
(LLUVIA FRIO NORMAL FUERTE NO)
(NUBLADO FRIO NORMAL FUERTE SI)
(SOLEADO TEMPLADO ALTA DEBIL NO)
(SOLEADO FRIO NORMAL DEBIL SI)
(LLUVIA TEMPLADO NORMAL DEBIL SI)
(SOLEADO TEMPLADO NORMAL FUERTE SI)
(NUBLADO TEMPLADO ALTA FUERTE SI)
(NUBLADO CALOR NORMAL DEBIL SI)
(LLUVIA TEMPLADO ALTA FUERTE NO))
```

Pero sería más útil identificar cada ejemplo por medio de una llave y poder acceder a sus atributos por nombre, por ejemplo, preguntar por el valor de día en el ejemplo número siete. Las siguientes funciones son la base para este trabajo:

```
1 (defun put-value (attr inst val)
2   (setf (get inst attr) val))
3
4 (defun get-value (attr inst)
5   (get inst attr))
```

Su uso es el siguiente:

```
CL-USER> (put-value 'nombre 'ej1 'alejandro)
ALEJANDRO
CL-USER> (get-value 'nombre 'ej1)
ALEJANDRO
CL-USER> (setq *ejemplos* nil)
NIL
CL-USER> (push 'ej1 *ejemplos*)
(EJ1)
CL-USER> (get-value 'nombre (car *ejemplos*))
ALEJANDRO
```

6.3. Clasificación a partir de un árbol de decisión

El procedimiento para clasificar un caso nuevo utilizando un árbol de decisión consiste en filtrar el ejemplo de manera descendente, hasta encontrar una hoja que corresponde a la clase buscada. Consideren el proceso de clasificación del siguiente caso: $\langle \text{Cielo} = \text{soleado}, \text{Temperatura} = \text{caliente}, \text{Humedad} = \text{alta}, \text{Viento} = \text{fuerte} \rangle$. Como el atributo *Cielo*, tiene el valor *soleado* en el ejemplar, éste es filtrado hacia abajo del árbol por la rama de la derecha. Como el atributo *Humedad*, tiene el valor *alta*, el ejemplo es filtrado nuevamente por rama de la derecha, lo cual nos lleva a la hoja que indica la clasificación del ejemplar: *Buen día para jugar tenis = no*. El algoritmo de clasificación se muestra en el algoritmo 1.

Algoritmo 1 El algoritmo clasifica, para árboles de decisión

```

1: function CLASIFICA(E: ejemplo, A: árbol)
2:   Clase  $\leftarrow$  valor-atributo(raíz(A),E);
3:   if es-hoja(raíz(A)) then
4:     return Clase
5:   else
6:     clasifica(E, sub-árbol(A,Clase));
7:   end if
8: end function

```

La implementación en Lisp de este algoritmo, dada la representación del árbol adoptada y las funciones de acceso definidas, es la siguiente:

```

1 (defun classify (instance tree)
2   (let* ((val (get-value (root tree) instance))
3         (branch (sub-tree tree val)))
4     (if (leaf branch) branch
5         (classify instance branch)))

```

La función `get-value` encuentra el valor de un atributo, en el caso que se está clasificando. Su implementación depende de como representemos el conjunto de ejemplos de entrenamiento. Si hemos alcanzado una hoja del árbol, el algoritmo regresa la hoja encontrada, en otro caso se procede recursivamente.

En general, un árbol de decisión representa una disyunción de conjunciones de restricciones en los posibles valores de los atributos de los casos. Cada rama que va de la raíz del árbol a una hoja, representa una conjunción de tales restricciones y el árbol mismo representa la disyunción de esas conjunciones. Por ejemplo, el árbol de la figura 6.1, puede expresarse como sigue:

$$\begin{aligned}
 & (\text{Cielo} = \text{soleado} \wedge \text{Humedad} = \text{normal}) \\
 \vee & (\text{Cielo} = \text{nublado}) \\
 \vee & (\text{Cielo} = \text{lluvia} \wedge \text{Viento} = \text{debil})
 \end{aligned}$$

6.4. El algoritmo básico de aprendizaje de árboles de decisión

La mayoría de los algoritmos para inferir árboles de decisión son variaciones de un algoritmo básico que emplea una búsqueda descendente (*top-down*) y egoísta (*greedy*) en el espacio de posibles árboles de decisión. La presentación de estos algoritmos se centra en ID3 y C4.5.

El algoritmo básico ID3 (Figura 2), construye el árbol de decisión de manera descendente, comenzando por preguntarse: Qué atributo debería ser colocado en la raíz del árbol? Para responder esta pregunta, cada atributo es evaluado usando un test estadístico para determinar que tan bien clasifica él solo los ejemplos de entrenamiento. El mejor atributo es seleccionado y colocado en la raíz del árbol. Una rama y su nodo correspondiente es entonces creada para cada valor posible del atributo en cuestión. Los ejemplos de entrenamiento son repartidos en los nodos descendentes de acuerdo al valor que tengan para el atributo de la raíz. El proceso entonces se repite con los ejemplos ya distribuidos, para seleccionar un atributo que será colocado en cada uno de los nodos generados. Generalmente, el algoritmo se detiene si los ejemplos de entrenamiento comparten el mismo valor para el atributo que está siendo probado. Sin embargo, otros criterios para finalizar la búsqueda son posibles: i) Cobertura mínima, el número de ejemplos cubiertos por cada nodo está por abajo de cierto umbral; ii) Pruebas de significancia estadística, usando χ^2 para probar si las distribuciones de las clases en los sub-árboles difiere significativamente. Puesto que el algoritmo lleva a cabo una búsqueda egoísta, sólo regresa un árbol de decisión aceptable, sin reconsiderar nunca las elecciones pasadas (*backtracking*). Por lo tanto el árbol computado puede no ser el óptimo para los ejemplos usados.

Algoritmo 2 Algoritmo para la inducción de árboles de decisión

```

1: function ID3(E: ejemplos, A: atributos, C: clase)
2:   AD  $\leftarrow$   $\emptyset$ ; Clase  $\leftarrow$  clase-mayoritaria(C,E);
3:   if E =  $\emptyset$  then return AD;
4:   else if misma-clase(E,C) then return AD  $\leftarrow$  Clase;
5:   else if A =  $\emptyset$  then return AD  $\leftarrow$  Clase;
6:   else
7:     Mejor-Partición  $\leftarrow$  mejor-partición(E, A);
8:     Mejor-Atributo  $\leftarrow$  primero(Mejor-Partición);
9:     AD  $\leftarrow$  Mejor-Atributo;
10:    for all Partición  $\in$  resto(Mejor-Partición) do
11:      Valor-Atributo  $\leftarrow$  primero(Partición);
12:      Sub-E  $\leftarrow$  resto(Partición);
13:      agregar-rama(AD, Valor-Atributo, ID3(Sub-E, {A \ Mejor-Atributo}, C));
14:    end for
15:  end if
16: end function

```

Antes de revisar la implementación de éste algoritmo, solucionemos la lectura de ejemplos de entrenamiento.

6.5. Cargando ejemplos de entrenamiento

Normalmente, los ejemplos de entrenamiento están almacenados en un archivo que Lisp no puede leer directamente, como una hoja de cálculo, una base de datos relacional o un archivo de texto con un formato definido. La primer decisión con respecto a los ejemplos de entrenamiento, es como serán cargados en Lisp.

6.5.1. Formatos de los archivos

En esta ocasión he decidido usar el formato ARFF usado por algunas herramientas de aprendizaje automático como Weka [24]. Escencialmente Weka usa un formato separado por comas y utiliza etiquetas para definir atributos y sus dominios (attribute, la clase (relation), comentarios (%) y el inicio de los datos (data). El conjunto de entrenamiento sobre jugar tenis quedaría representado en este formato como un archivo con extensión `.arff`:

```

1 @RELATION jugar-tenis
2
3 @ATTRIBUTE cielo {soleado,nublado,lluvia}
4 @ATTRIBUTE temperatura {calor,templado,frio}
5 @ATTRIBUTE humedad {alta,normal}
6 @ATTRIBUTE viento {debil,fuerte}
7 @ATTRIBUTE jugar-tenis {si,no}
8
9 @DATA
10
11 soleado, calor, alta, debil, no
12 soleado, calor, alta, fuerte, no
13 nublado, calor, alta, debil, si
14 lluvia, templado, alta, debil, si
15 lluvia, frio, normal, debil, si
16 lluvia, frio, normal, fuerte, no
17 nublado, frio, normal, fuerte, si
18 soleado, templado, alta, debil, no
19 soleado, frio, normal, debil, si
20 lluvia, templado, normal, debil, si
21 soleado, templado, normal, fuerte, si
22 nublado, templado, alta, fuerte, si
23 nublado, calor, normal, debil, si
24 lluvia, templado, alta, fuerte, no

```

También sería deseable que nuestro programa pudiera cargar conjuntos de entrenamiento en formato CSV, usados ampliamente. Aquí el archivo anterior luciría como:

```

1 cielo, temperatura, humedad, viento, jugar-tenis
2 soleado, calor, alta, debil, no
3 soleado, calor, alta, fuerte, no
4 nublado, calor, alta, debil, si
5 lluvia, templado, alta, debil, si
6 ...

```

6.5.2. Ambiente de aprendizaje

Primero necesitamos declarar algunas variables globales, para identificar datos, atributos, ejemplos, etc. Esto configura nuestro ambiente de aprendizaje.

```

1  ;; Global variables
2
3  (defvar *examples* nil "The training set")
4  (defvar *attributes* nil "The attributes of the problem")
5  (defvar *data* nil "The values of the attributes of all *examples*")
6  (defvar *domains* nil "The domain of the attributes")
7  (defvar *target* nil "The target concept")
8  (defvar *trace* nil "Trace the computations")
9  (defvar *root* nil "Temporal hook for displaying the tree")

```

las variables son auto explicativas.

6.5.3. Lectura de archivos

Ahora tenemos dos opciones, leer el archivo usando alguna utilidad del sistema operativo en el que estamos ejecutando Lisp, por ejemplo `grep` en Unix; ó programar directamente la lectura de archivos. Primero veremos la versión programada enteramente programada en Lisp.

```

1  (defun read-lines-from-file (file)
2    (remove-if (lambda (x) (equal x ""))
3              (with-open-file (in file)
4                (loop for line = (read-line in nil 'end)
5                      until (eq line 'end) collect line))))

```

La función definida permite obtener una lista de cadenas de caracteres, donde cada cadena corresponde con una línea del archivo. Algunas de las funciones definidas hacen uso de `split-sequence` que está definida en una librería no estándar de Lisp. Esto significa que ustedes tendrán que instalar la librería antes de poder compilar las definiciones listadas a continuación. La próxima sección aborda la instalación y uso de librerías.

```

CL-USER> (read-lines-from-file "tenis.arff")
("@RELATION jugar-tenis" "@ATTRIBUTE cielo {soleado,nublado,lluvia}" "
 @ATTRIBUTE temperatura {calor,templado,frio}" "@ATTRIBUTE humedad {alta,
normal}" "@ATTRIBUTE viento {debil,fuerte}" "@ATTRIBUTE jugar-tenis {si,no
}" "soleado, calor, alta, debil, no" "soleado, calor, alta, fuerte, no" "
nublado, calor, alta, debil, si" "lluvia, templado, alta, debil, si" "
lluvia, frio, normal, debil, si" "lluvia, frio, normal, fuerte, no" "
nublado, frio, normal, fuerte, si" "soleado, templado, alta, debil, no" "
soleado, frio, normal, debil, si" "lluvia, templado, normal, debil, si" "
soleado, templado, normal, fuerte, si" "nublado, templado, alta, fuerte,
si" "nublado, calor, normal, debil, si" "lluvia, templado, alta, fuerte,
no")

```

Ahora necesitamos manipular la cadena obtenida para instanciar adecuadamente las variables de nuestro ambiente de aprendizaje. Para el caso de los archivos ARFF estás son las funciones básicas:

```

1 (defun arff-get-target (lines)
2   "It extracts the value for *target* from the lines of a ARFF file"
3   (read-from-string
4     (cadr (split-sequence
5           #\Space
6           (car (remove-if-not
7                 (lambda (x) (or (string-equal "@r" (subseq x 0 2))
8                               (string-equal "@R" (subseq x 0 2))))
9                 lines))))))
10
11 (defun arff-get-data (lines)
12   "It extracts the value for *data* from the lines of a ARFF file"
13   (mapcar #'(lambda (x)
14             (mapcar #'read-from-string
15                   (split-sequence #\, x)))
16         (remove-if
17         (lambda (x) (string-equal "@" (subseq x 0 1)))
18         lines))
19
20 (defun arff-get-attrs-doms (lines)
21   " It extracts the list (attributes domains) from an ARFF file"
22   (mapcar #'(lambda (x)
23             (list (read-from-string (car x))
24                 (mapcar #'read-from-string
25                       (split-sequence
26                        #\,
27                        (remove-if (lambda (x)
28                                  (or (string-equal "{" x)
29                                      (string-equal "}" x)))
30                                  (cadr x))))))
31         (mapcar #'(lambda (x)
32                   (cdr (split-sequence
33                         #\Space x)))
34               (remove-if-not
35               (lambda (x)
36                 (or (string-equal "@a" (subseq x 0 2))
37                     (string-equal "@A" (subseq x 0 2))))
38               lines))))

```

Así, podemos extraer la clase y los datos del archivo ARFF como se muestra a continuación:

```

CL-ID3> (arff-get-target (read-lines-from-file "tenis.arff"))
JUGAR-TENIS
11
CL-ID3> (arff-get-data (read-lines-from-file "tenis.arff"))
((SOLEADO CALOR ALTA DEBIL NO) (SOLEADO CALOR ALTA FUERTE NO) (NUBLADO CALOR
ALTA DEBIL SI) (LLUVIA TEMPLADO ALTA DEBIL SI) (LLUVIA FRIO NORMAL DEBIL
SI) (LLUVIA FRIO NORMAL FUERTE NO) (NUBLADO FRIO NORMAL FUERTE SI) (
SOLEADO TEMPLADO ALTA DEBIL NO) (SOLEADO FRIO NORMAL DEBIL SI) (LLUVIA
TEMPLADO NORMAL DEBIL SI) (SOLEADO TEMPLADO NORMAL FUERTE SI) (NUBLADO
TEMPLADO ALTA FUERTE SI) (NUBLADO CALOR NORMAL DEBIL SI) (LLUVIA TEMPLADO
ALTA FUERTE NO))

```

Evidentemente, necesitamos implementar versiones de estas funciones para el caso de que el archivo de entrada esté en formato CSV.

```

1 (defun csv-get-target (lines)
2   "It extracts the value for *target* from the lines of a CSV file"
3   (read-from-string
4     (car (last (split-sequence #\, (car lines))))))
5
6 (defun csv-get-data (lines)
7   "It extracts the value for *data* from the lines of a CSV file"
8   (mapcar #'(lambda(x)
9             (mapcar #'read-from-string
10                  (split-sequence #\, x)))
11         (cdr lines)))
12
13 (defun csv-get-attrs-doms (lines)
14   "It extracts the list (attributes domains) from an CSV file"
15   (labels ((csv-get-values (attrs data)
16            (loop for a in attrs collect
17                  (remove-duplicates
18                    (mapcar #'(lambda(l)
19                              (nth (position a attrs) l))
20                              data))))))
21     (let* ((attrs (mapcar #'read-from-string
22                          (split-sequence #\, (car lines))))
23            (data (csv-get-data lines))
24            (values (csv-get-values attrs data)))
25       (mapcar #'list attrs values)))

```

La función principal para cargar un ambiente de aprendizaje es la siguiente:

```

1 (defun load-file (file)
2   "It initializes the learning setting from FILE"
3   (labels ((get-examples (data)
4            (loop for d in data do
5                  (let ((ej (gensym "ej")))
6                    (setf *examples* (cons ej *examples*))
7                    (loop for attrib in *attributes*
8                          as v in d do
9                            (put-value attrib ej v))))))
10     (if (probe-file file)
11         (let ((file-ext (car (last (split-sequence #\. file))))
12             (file-lines (read-lines-from-file file)))
13           (reset)
14           (cond
15             ((equal file-ext "arff")
16              (let ((attrs-doms (arff-get-attrs-doms file-lines))
17                  (setf *attributes* (mapcar #'car attrs-doms)
18                        *domains* (mapcar #'cadr attrs-doms)
19                        *target* (arff-get-target file-lines)
20                        *data* (arff-get-data file-lines))
21                (get-examples *data*)
22                (format t "Training set initialized after ~s.~%" file)))
16             ((equal file-ext "csv")
17              (let ((attrs-doms (csv-get-attrs-doms file-lines))
18                  (setf *attributes* (mapcar #'car attrs-doms)
19                        *domains* (mapcar #'cadr attrs-doms)
20                        *target* (csv-get-target file-lines)
21                        *data* (csv-get-data file-lines))
22                (get-examples *data*)
23                (format t "Training set initialized after ~s.~%" file)))
24              (t (error "File's ~s extension can not be determined." file)))
25              (error "File ~s does not exist.~%" file)))

```

La función `reset` reinicia los valores de las variables globales que configuran el ambiente de aprendizaje:

```

1 (defun reset ()
2   (setf *data* nil
3         *examples* nil
4         *target* nil
5         *attributes* nil
6         *domains* nil
7         *root* nil
8         *gensym-counter* 1)
9   (format t "The ID3 setting has been reset.~%"))

```

En la distribución del sistema todas estas definiciones se encuentran en el archivo `cl-id3-load.lisp`. Con el código cargado en Lisp, podemos hacer lo siguiente:

```

CL-ID3> (load-file "tenis.arff")
The ID3 setting has been reset.
Training set initialized after "tenis.arff".
NIL
CL-ID3> *target*
JUGAR-TENIS
CL-ID3> *attributes*
(CIELO TEMPERATURA HUMEDAD VIENTO JUGAR-TENIS)
CL-ID3> *examples*
(#:|ej14| #:|ej13| #:|ej12| #:|ej11| #:|ej10| #:|ej9| #:|ej8| #:|ej7| #:|ej6|
 #:|ej5| #:|ej4| #:|ej3| #:|ej2| #:|ej1|)

```

Como pueden observar, el ambiente de aprendizaje ha sido inicializado con los datos guardados en `tenis.arff`. Lo mismo sucedería para `tenis.csv`.

6.6. Librerías ASDF instalables

Los programadores suelen quejarse de la aproximación usada por Lisp para las librerías. De hecho, se suele asumir que no existen librerías en Lisp. Aunque esto último es falso, lo cierto es que las distribuciones de Lisp no vienen acompañadas de un conjunto de librerías estandarizadas, ni existe un repositorio unificado de ellas.

ASDF y `ASDF-install` ayudan a mantener repositorios de librerías asociados a nuestra instalación de Lisp, así como su localización en la web. El índice de librerías instalables por este método se encuentra en: <http://www.cliki.net>. Un buen tutorial está disponible en: <http://common-lisp.net/project/asdf-install/tutorial/introduction.html>.

Algunas distribuciones de Lisp ya incluyen ASDF, por ejemplo SBCL, Clozure y Lispworks 6.0. Si ese no es el caso, habrá que instalar ASDF antes de instalar `ASDF-install`. Para ello hay que crear un directorio donde colocaremos el código de ASDF, compilar el código y cargarlo en nuestro ambiente Lisp mediante un archivo de inicialización.

En mi caso decidí instalar ASDF `ASDF-install` en una carpeta `lisp` donde guardaré todo sobre Lisp en mi Macbook Pro. Como Lisp utilizaré Lispworks 5.1.2 Professional Edition. El procedimiento de instalación que seguiremos es igual para cualquier sistema UNIX y similares.

6.6.1. Instalación de ASDF

En una terminal:

```
clea:~ aguerra$ mkdir lisp
clea:~ aguerra$ cd lisp
clea:lisp aguerra$ curl http://common-lisp.net/project/asdf/asdf.lisp -o asdf.
lisp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload  Total  Spent  Left  Speed
100 85928  100 85928    0     0  61836      0  0:00:01  0:00:01 --:--:-- 71487
clea:lisp aguerra$ ls
asdf.lisp
```

Luego, desde una consola de Lisp, compilar el archivo `asdf.lisp`:

```
CL-USER> (change-directory "~/lisp")
#P"/Users/aguerra/lisp/"
CL-USER> (compile-file "asdf.lisp")
... muchas líneas de salida
T
```

Nuestro directorio debe incluir ahora los siguientes archivos:

```
clea:lisp aguerra$ ls
asdf.lisp      asdf.xfasl
clea:lisp aguerra$
```

Es necesario crear una estructura de directorios donde las librerías ASDF instalables serán almacenadas permanentemente. En mi caso:

```
clea:lisp aguerra$ mkdir .asdf-install-dir
clea:lisp aguerra$ mkdir .asdf-install-dir/site
clea:lisp aguerra$ mkdir .asdf-install-dir/systems
clea:lisp aguerra$
```

Finalmente debo incluir al principio de mi archivo de inicialización `.lispworks` las siguientes líneas:

```
1 #-:asdf (load "/Users/aguerra/lisp/asdf")
2 (pushnew "/Users/aguerra/.asdf-install-dir/systems/" asdf:*central-registry*
3   :test #'equal)
```

esto hace que ASDF sepa donde buscar las librerías instaladas en el sistema.

6.6.2. Instalación de ASDF-install

El repositorio web de librerías ASDF instalables, es gestionado por la librería `ASDF-install`. Como es necesario que esta librería esté instalada para bajar otras

librerías, su instalación es manual. A continuación muestro su instalación desde una terminal, con el directorio de trabajo en `/Users/aguerra/lisp`:

```
clea:lisp aguerra$ curl http://common-lisp.net/project/asdf-install/
  asdf-install_latest.tar.gz -o asdf-install.tar.gz
  % Total    % Received % Xferd Average Speed   Time    Time     Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 84270  100 84270    0     0  65710      0  0:00:01  0:00:01 --:--:-- 77311
clea:lisp aguerra$ gunzip asdf-install.tar.gz
clea:lisp aguerra$ tar -xvf asdf-install.tar
  asdf-install/
  asdf-install/asdf-install/
  asdf-install/asdf-install/asdf-install.asd
  asdf-install/asdf-install/changes.text
  asdf-install/asdf-install/conditions.lisp
  ...
clea:lisp aguerra$ ls -l
total 1112
drwxr-xr-x  7 aguerra  staff    238 Dec 26  2007 asdf-install
-rw-r--r--  1 aguerra  staff  286720 Jan 12 15:13 asdf-install.tar
-rw-r--r--  1 aguerra  staff   85928 Jan 12 14:39 asdf.lisp
-rw-r--r--  1 aguerra  staff  194421 Jan 12 14:50 asdf.xfasl
```

El directorio `asdf-install` contiene un subdirectorio del mismo nombre, donde se ubica el archivo `asdf-install.asd`; eso es, la definición del sistema en formato ASDF. Es necesario que este archivo este disponible en el registro central de ASDF, para ello creamos una liga del repositorio a este archivo:

```
clea:lisp aguerra$ cd
clea:~ aguerra$ cd .asdf-install-dir/systems/
clea:systems aguerra$ ln -s /Users/aguerra/lisp/asdf-install/asdf-install/
  asdf-install.asd .
clea:systems aguerra$
```

No olviden el punto al final de la tercera línea. Verifiquen que la herramienta `tar` del sistema operativo sea la versión GNU. OS X Snow Leopard instala BSD `tar` por defecto, por lo que es necesario cambiar eso, ya que `ASDF-install` depende de la salida ofrecida por la versión GNU:

```
clea:~ aguerra$ cd /usr/bin; sudo ln -fs gnutar tar && /usr/bin/tar --version
Password:
tar (GNU tar) 1.17
Copyright (C) 2007 Free Software Foundation, Inc.
...
clea:usr/bin aguerra$
```

Finalmente agregamos una línea al final de nuestras modificaciones al archivo de inicialización `.lispworks`, para que `ASDF-install` esté disponible al momento de arrancar nuestro `Lispworks`:

```
1 | #-:asdf-install (asdf:operate 'asdf:load-op :asdf-install)
```

6.6.3. *Uso de librerías ASDF instalables*

Para instalar la librería `split-sequence` procedemos de la siguiente manera:

```
CL-USER> (asdf-install:install :split-sequence)
Install where?
1) System-wide install:
   System in /usr/local/asdf-install/site-systems/
   Files in /usr/local/asdf-install/site/
2) Personal installation:
   System in /Users/aguerra/.asdf-install-dir/systems/
   Files in /Users/aguerra/.asdf-install-dir/site/
0) Abort installation.
--> 2

;;; ASDF-INSTALL: Downloading 2601 bytes from http://ftp.linux.org.uk/pub/lisp/
experimental/cclan/split-sequence.tar.gz to /Users/aguerra/asdf-install-0.
asdf-install-tmp ...
;;; ASDF-INSTALL: Downloading 189 bytes from http://ftp.linux.org.uk/pub/lisp/
experimental/cclan/split-sequence.tar.gz.asc to /Users/aguerra/
asdf-install-1.asdf-install-tmp ...

"pgp: Firmado el Wed Jun  4 12:00:19 2003 CDT usando clave DSA ID 52D68DF2"
"[GNUPG:] SIG_ID R+qba5kYVsu0r6GQ4vjp0WCHs50 2003-06-04 1054746019"
"[GNUPG:] GOODSIG 84C5E27852D68DF2 Christophe Rhodes <csr21@cam.ac.uk>"
"pgp: Firma correcta de \"Christophe Rhodes <csr21@cam.ac.uk>\""
"[GNUPG:] VALIDSIG B36B91C51835DB9BFBAB735B84C5E27852D68DF2 2003-06-04
1054746019 0 3 0 17 2 00 B36B91C51835DB9BFBAB735B84C5E27852D68DF2"
"[GNUPG:] TRUST_UNDEFINED"
"pgp: ATENCION: ¡Esta clave no está; certificada por una firma de confianza!"
"pgp:          No hay indicios de que la firma pertenezca al propietario."
"Huellas dactilares de la clave primaria: B36B 91C5 1835 DB9B FBAB 735B 84C5
E278 52D6 8DF2"
;;; ASDF-INSTALL: Installing SPLIT-SEQUENCE in /Users/aguerra/.asdf-install-dir
/site/, /Users/aguerra/.asdf-install-dir/systems/
"ln -s \"/Users/aguerra/.asdf-install-dir/site/split-sequence/split-sequence.
asd\" \" /Users/aguerra/.asdf-install-dir/systems/split-sequence.asd\""
;;; ASDF-INSTALL: Found system definition: /Users/aguerra/.asdf-install-dir/
site/split-sequence/split-sequence.asd
;;; ASDF-INSTALL: Loading system ASDF-INSTALL::SPLIT-SEQUENCE via ASDF.
(ASDF-INSTALL::SPLIT-SEQUENCE)
```

La librería ha sido cargada, compilada y copiada al registro central de ASDF. Si no se tiene instalado GNUPG, Lisp se quejará argumentando que no hay una firma segura que valide la operación de instalación. Se puede seleccionar la opción de continuar la instalación sin verificar la firma digital de la librería descargada. Aunque, lo mejor es instalar GNUPG.

Para usar la librería instalada podemos invocarla desde la consola Lisp:

```
CL-USER> (asdf:operate 'asdf:load-op :split-sequence)
#<ASDF:LOAD-OP NIL 200A68F7>
```

Y usarla en nuestros programas:

```
CL-USER> (split-sequence:split-sequence #\, "cielo,temperatura,humedad,viento")
("cielo" "temperatura" "humedad" "viento")
32
```

6.6.4. Definiendo una librería ASDF: cl-id3

En la medida que vayamos completando nuestra implementación de ID3, el código se irá haciendo más grande. Será conveniente separarlo en varios archivos y definir las dependencias de compilación entre estos usando ASDF. El siguiente listado corresponde al archivo `cl-id3.asd` incluido en la distribución final de nuestro programa.

```

1 (asdf:defsystem :cl-id3
2   :depends-on (:split-sequence)
3   :components ((:file "cl-id3-package")
4                 (:file "cl-id3-algorithm"
5                       :depends-on ("cl-id3-package"))
6                 (:file "cl-id3-load"
7                       :depends-on ("cl-id3-package"
8                                   "cl-id3-algorithm")))
9                 (:file "cl-id3-classify"
10                      :depends-on ("cl-id3-package"
11                                   "cl-id3-algorithm"
12                                   "cl-id3-load"))
13                (:file "cl-id3-cross-validation"
14                      :depends-on ("cl-id3-package"
15                                   "cl-id3-algorithm"
16                                   "cl-id3-load"
17                                   "cl-id3-classify"))
18                (:file "cl-id3-gui"
19                      :depends-on ("cl-id3-package"
20                                   "cl-id3-algorithm"
21                                   "cl-id3-load"
22                                   "cl-id3-classify"
23                                   "cl-id3-cross-validation"))))

```

Esta definición de sistema establece que nuestra librería `cl-id3` depende de la librería `split-sequence`, de forma que ésta última se cargará en Lisp antes de intentar compilar y cargar nuestras fuentes. El orden en que nuestras fuentes son compiladas y cargadas se establece en los `:depends-on` de la definición.

6.7. Paquetes: cl-id3

Observen que la función `split-sequence` está definida dentro del paquete `split-sequence` de forma que si queremos llamar a esta función desde la consola de Lisp, debemos indicar el paquete al que pertenece. Esto es necesario porque por defecto estamos ubicados en el paquete `cl-user`.

Es conveniente definir un paquete para nuestra aplicación de forma que las funciones relevantes no se confundan con las definidas en el paquete `cl-user`. A continuación listamos el archivo `cl-id3-package.lisp`:

```

1 ;; cl-id3-package
2 ;; The package for cl-id3
3
4 (defpackage :cl-id3
5   (:use :cl :capi :split-sequence))

```

```

6  (:export :load-file
7      :induce
8      :print-tree
9      :classify
10     :classify-new-instance
11     :cross-validation
12     :gui))

```

Esta definición le indica a Lisp que el paquete `cl-id3` hace uso de los paquetes `common-lisp`, `capi` y `split-sequence`, por lo que no será necesario indicarlos al invocar sus funciones en nuestro código (observen que las funciones de carga de archivos llaman a `split-sequence`, sin indicar a que paquete pertenece). Los símbolos definidos bajo `:export` son visibles desde otros paquetes.

Es necesario que el resto de nuestros archivos fuente, incluyan como primer línea lo siguiente:

```

1  (in-package :cl-id3)

```

Una vez que el sistema `cl-id3` es compilado y cargado en Lisp, se puede usar como se muestra a continuación:

```

CL-USER> (cl-id3:load-file "tenis.arff")
The ID3 setting has been reset.
Training set initialized after "tenis.arff".
NIL
CL-USER> (cl-id3:induce)
(CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI) (LLUVIA (VIENTO (
FUERTE NO) (DEBIL SI))))

```

Volvamos a la definición del algoritmo ID3.

6.8. ¿Qué atributo es el mejor clasificador?

La decisión central de ID3 consiste en seleccionar qué atributo colocará en cada nodo del árbol de decisión. En el algoritmo presentado, esta opción la lleva a cabo la función `mejor-partición`, que toma como argumentos un conjunto de ejemplos de entrenamiento y un conjunto de atributos, regresando la partición inducida por el atributo, que sólo, clasifica mejor los ejemplos de entrenamiento.

6.8.1. Particiones

Como pueden observar en la descripción del algoritmo ID3 (Algoritmo 2, página 68), una operación común sobre el conjunto de entrenamiento es la de partición con respecto a algún atributo. La idea es tener una función que tome un atributo y un conjunto de ejemplos y los particione de acuerdo a los valores observados del atributo, por ejemplo:

```

1 CL-USER> (in-package :cl-id3)
2 #<The CL-ID3 package, 148/512 internal, 7/16 external>
3 CL-ID3> (load-file "tenis.arff")
4 The ID3 setting has been reset.
5 Training set initialized after "tenis.arff".
6 NIL
7 CL-ID3> (get-partition 'temperatura *examples*)
8 (TEMPERATURA (FRIO #:|ej5| #:|ej6| #:|ej7| #:|ej9|) (CALOR #:|ej1| #:|ej2| #:|
   ej3| #:|ej13|) (TEMPLADO #:|ej4| #:|ej8| #:|ej10| #:|ej11| #:|ej12| #:|
   ej14|))

```

Lo que significa que el atributo *temperatura* tiene tres valores diferentes en el conjunto de entrenamiento: *frío*, *calor*, y *templado*. Los ejemplos 5,6,7 y 9 tienen como valor del atributo *temperatura*= *frío*, etc. La definición de la función `get-partition` es como sigue:

```

1 (defun get-partition (attrib examples)
2   "It gets the partition induced by ATTRIB in EXAMPLES"
3   (let (result vlist v)
4     (loop for e in examples do
5       (setq v (get-value attrib e))
6       (if (setq vlist (assoc v result))
7           ;; value v existed, the example e is added
8           ;; to the cdr of vlist
9           (rplacd vlist (cons e (cdr vlist)))
10          ;; else a pair (v e) is added to result
11          (setq result (cons (list v e) result))))
12   (cons attrib result))

```

el truco está en el `if` de la línea 6, que determina si el valor del atributo en el ejemplo actual es un nuevo valor o uno ya existente. Si se trata de un nuevo valor lo inserta en `result` como una lista (`valor ejemplo`). Si ya existía, `rplacd` se encarga de reemplazar el `cdr` de la lista (`valor ejemplo`) existente, agregando el nuevo ejemplo: (`valor ejemplo ejemplo-nuevo`)!

Necesitaremos una función *best-partition* que encuentre el atributo que mejor separa los ejemplos de entrenamiento de acuerdo a la clase buscada ¿En qué consiste una buena medida cuantitativa de la bondad de un atributo? Para contestar a esta cuestión, definiremos una propiedad estadística llamada *ganancia de información*.

6.8.2. Entropía y ganancia de información

Una manera de cuantificar la bondad de un atributo en este contexto, consiste en considerar la cantidad de información que proveerá este atributo, tal y como éste es definido en teoría de información por Claude E. Shannon [23]. Un bit de información es suficiente para determinar el valor de un atributo booleano, por ejemplo, *si/no*, *verdader/falso*, *1/0*, etc., sobre el cual no sabemos nada. En general, si los posibles valores del atributo v_i , ocurren con probabilidades $P(v_i)$, entonces en contenido de información, o entropía, E de la respuesta actual está dado por:

$$E(P(v_1), \dots, P(v_n)) = \sum_{i=1}^n -P(v_i) \log_2 P(v_i)$$

Consideren nuevamente el caso booleano, aplicando esta ecuación a un volado con una moneda confiable, tenemos que la probabilidad de obtener aguilá o sol es de $1/2$ para cada una:

$$E\left(\frac{1}{2}, \frac{1}{2}\right) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1$$

Ejecutar el volado nos provee 1 bit de información, de hecho, nos provee la clasificación del experimento: si fue aguilá o sol. Si los volados los ejecutamos con una moneda cargada que da 99% de las veces sol, entonces $E(1/100, 99/100) = 0,08$ bits de información, menos que en el caso de la moneda justa, porque ahora tenemos más evidencia sobre el posible resultado del experimento. Si la probabilidad de que el volado de sol es del 100%, entonces $E(0, 1) = 0$ bits de información, ejecutar el volado no provee información alguna. La gráfica de la función de entropía se muestra en la figura 6.2.

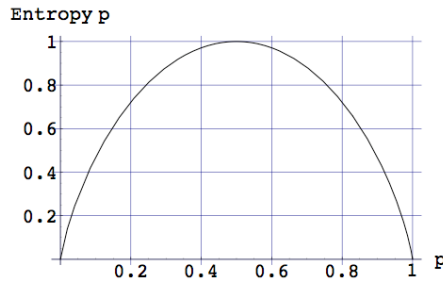


Figura 6.2 Gráfica de la función entropía para clasificaciones booleanas.

Consideren nuevamente los ejemplos de entrenamiento del cuadro 6.1. De 14 ejemplos, 9 son positivos (si es un buen día para jugar tenis) y 5 son negativos. La entropía de este conjunto de entrenamiento es:

$$E\left(\frac{9}{14}, \frac{5}{14}\right) = 0,940$$

Si todos los ejemplos son positivos o negativos, por ejemplo, pertenecen todos a la misma clase, la entropía será 0. Una posible interpretación de ésto, es considerar la entropía como una medida de ruido o desorden en los ejemplos. Definimos la *ganancia de información* (GI) como la reducción de la entropía causada por particionar un conjunto de entrenamiento S , con respecto a un atributo a :

$$\text{Ganancia}(S, a) = E(S) - \sum_{v \in a} \frac{|S_v|}{|S|} E(S_v)$$

Observen que el segundo término de *Ganancia*, es la entropía con respecto al atributo *a*. Al utilizar esta medida en ID3, sobre los ejemplos del cuadro 6.1, deberíamos obtener algo como:

```
1 | CL-USER> (CL-ID3> (best-partition (remove *target* *attributes*)
2 | *examples*)
3 | (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|) (NUBLADO #:|ej3| #:|
   | ej7| #:|ej12| #:|ej13|) (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|)
   | )
```

Esto indica que el atributo con mayor ganancia de información fue *cielo*, de ahí que esta parte del algoritmo genera la partición de los ejemplos de entrenamiento con respecto a este atributo. Si particionamos recursivamente los ejemplos que tienen el atributo *cielo* = *soleado*, obtendríamos:

```
1 | (HUMEDAD (NORMAL #:|ej9| #:|ej11|)
2 | (ALTA #:|ej1| #:|ej2| #:|ej8|))
```

Lo cual indica que en el nodo debajo de *soleado* deberíamos incluir el atributo *humedad*. Todos los ejemplos con *humedad* = *normal*, tienen valor *si* para el concepto objetivo. De la misma forma, todos los ejemplos con valor *humedad* = *alta*, tiene valor *no* para el concepto objetivo. Así que ambas ramas descendiendo de nodo *humedad*, llevarán a clases terminales de nuestro problema de aprendizaje. El algoritmo terminará por construir el árbol de la figura 6.1.

Esto nos da las pistas necesarias para programar ID3. Primero, tenemos la función que computa entropía:

```
1 | (defun entropy (examples attrib)
2 | "It computes the entropy of EXAMPLES with respect to an ATTRIB"
3 | (let ((partition (get-partition attrib examples))
4 | (number-of-examples (length examples)))
5 | (apply #' +
6 | (mapcar #' (lambda (part)
7 | (let* ((size-part (count-if #'atom
8 | (cdr part)))
9 | (proportion
10 | (if (eq size-part 0) 0
11 | (/ size-part
12 | number-of-examples))))
13 | (* -1.0 proportion (log proportion 2))))
14 | (cdr partition))))
```

Si queremos ahora saber la entropía del conjunto de **examples** con respecto a la clase *jugar-tenis*, tenemos que:

```
1 | CL-USER> (entropy *examples* 'jugar-tenis)
2 | 0.9402859
```

Ahora, para computar ganancia de información, definimos:

```
1 | (defun information-gain (examples attribute)
2 | "It computes information-gain for an ATTRIBUTE in EXAMPLES"
```

```

3 (let ((parts (get-partition attribute examples))
4       (no-examples (count-if #'atom examples)))
5   (- (entropy examples *target*)
6      (apply #'+
7             (mapcar
8              #'(lambda(part)
9                  (let* ((size-part (count-if #'atom
10                                     (cdr part)))
11                        (proportion (if (eq size-part 0) 0
12                                       (/ size-part
13                                           no-examples))))
14                    (* proportion (entropy (cdr part) *target*)))
15                (cdr parts))))))

```

de forma que la ganancia de información del atributo `cielo`, con respecto a la clase `jugar-tenis`, puede obtenerse de la siguiente manera:

```

1 CL-USER> (information-gain *examples* 'cielo 'jugar-tenis)
2 0.24674976

```

Ahora podemos implementar la función para encontrar la mejor partición con respecto a la ganancia de información:

```

1 (defun best-partition (attributes examples)
2   "It computes one of the best partitions induced by ATTRIBUTES over EXAMPLES"
3   (let* ((info-gains (loop for attrib in attributes collect
4                           (let ((ig (information-gain examples attrib))
5                                 (p (get-partition attrib examples)))
6                               (when *trace*
7                                 (format t "Partición inducida por el atributo ~s
8                                         :~s~s%"
9                                         attrib p)
10                                (format t "Ganancia de información: ~s~s%"
11                                        ig)
12                                (list ig p))))
13         (best (cadar (sort info-gains #'(lambda(x y) (> (car x) (car y))))))
14         (when *trace* (format t "Best partition: ~s~s-----~s%" best)
15         best))

```

Si queremos encontrar la mejor partición inicial, tenemos:

```

1 CL-ID3> (best-partition (remove *target* *attributes*)
2              *examples*)
3 (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|) (NUBLADO #:|ej3| #:|
4         ej7| #:|ej12| #:|ej13|) (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|)
5 )

```

Si queremos más información sobre cómo se obtuvo esta partición, podemos usar la opción de `(setf *trace* t)`:

```

1 CL-ID3> (setf *trace* t)
2 T
3 CL-ID3> (best-partition (remove *target* *attributes*)
4              *examples*)
5 Partición inducida por el atributo CIELO:
6 (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|) (NUBLADO #:|ej3| #:|
7         ej7| #:|ej12| #:|ej13|) (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|)
8 )

```

```

7  Ganancia de información: 0.2467497
8  Partición inducida por el atributo TEMPERATURA:
9  (TEMPERATURA (FRIO #:lej5| #:lej6| #:lej7| #:lej9|) (CALOR #:lej1| #:lej2| #:|
   ej3| #:lej13|) (TEMPLADO #:lej4| #:lej8| #:lej10| #:lej11| #:lej12| #:|
   ej14|))
10 Ganancia de información: 0.029222489
11 Partición inducida por el atributo HUMEDAD:
12 (HUMEDAD (NORMAL #:lej5| #:lej6| #:lej7| #:lej9| #:lej10| #:lej11| #:lej13|) (
   ALTA #:lej1| #:lej2| #:lej3| #:lej4| #:lej8| #:lej12| #:lej14|))
13 Ganancia de información: 0.15183544
14 Partición inducida por el atributo VIENTO:
15 (VIENTO (DEBIL #:lej1| #:lej3| #:lej4| #:lej5| #:lej8| #:lej9| #:lej10| #:lej13
   |) (FUERTE #:lej2| #:lej6| #:lej7| #:lej11| #:lej12| #:lej14|))
16 Ganancia de información: 0.048126936
17 Best partition: (CIELO (SOLEADO #:lej1| #:lej2| #:lej8| #:lej9| #:lej11|) (
   NUBLADO #:lej3| #:lej7| #:lej12| #:lej13|) (LLUVIA #:lej4| #:lej5| #:lej6|
   #:lej10| #:lej14|))
18 -----
19 (CIELO (SOLEADO #:lej1| #:lej2| #:lej8| #:lej9| #:lej11|) (NUBLADO #:lej3| #:|
   ej7| #:lej12| #:lej13|) (LLUVIA #:lej4| #:lej5| #:lej6| #:lej10| #:lej14|)
   )

```

id3 llama recursivamente a best-partition:

```

1  (defun id3 (examples attribs)
2  "It induces a decision tree running id3 over EXAMPLES and ATTRIBS)"
3  (let ((class-by-default (get-value *target*
4  (car examples))))
5
6  (cond
7  ;; Stop criteria
8  ((same-class-value-p *target*
9  class-by-default examples) class-by-default)
10
11  ;; Failure
12  ((null attribs) (target-most-common-value examples))
13  ;; Recursive call
14  (t (let ((partition (best-partition attribs
15  examples)))
16  (cons (first partition)
17  (loop for branch in (cdr partition) collect
18  (list (first branch)
19  (id3 (cdr branch)
20  (remove (first partition)
21  attribs))))))))))
22
23 (defun same-class-value-p (attrib value examples)
24 "Do all EXAMPLES have the same VALUE for a given ATTRIB ?"
25 (every #'(lambda(e)
26 (eq value
27 (get-value attrib e)))
28 examples))
29
30 (defun target-most-common-value (examples)
31 "It gets the most common value for *target* in EXAMPLES"
32 (let ((domain (get-domain *target*)))
33 (values (mapcar #'(lambda(x) (get-value *target* x))
34 examples)))
35 (caar (sort (loop for v in domain collect
36 (list v (count v values)))
37 #'(lambda(x y) (>= (cadr x)
38 (cadr y))))))
39
40 (defun get-domain (attribute)
41 "It gets the domain of an ATTRIBUTE"
42 (nth (position attribute *attributes*)
43 *domains*))

```

La implementación del algoritmo termina con una pequeña función de interfaz, para ejecutar `id3` sobre el ambiente de aprendizaje por defecto. Aprovecho esta función para verificar si la clase está incluida en el archivo ARFF, ya que WEKA puede eliminar ese atributo del archivo. El símbolo `induce` es exportado por el paquete `cl-id3`:

```

1 (defun induce (&optional (examples *examples*))
2   "It induces the decision tree using learning setting"
3   (when (not (member *target* *attributes*))
4     (error "The target is defined incorrectly: Maybe Weka modified your ARFF"))
5   (id3 examples (remove *target* *attributes*)))

```

De forma que para construir el árbol de decisión ejecutamos:

```

CL-ID3> (induce)
(CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI) (LLUVIA (VIENTO (
FUERTE NO) (DEBIL SI))))

```

Podemos definir una función para imprimir el árbol de manera más amigable:

```

1 (defun print-tree (tree &optional (depth 0))
2   (mytab depth)
3   (format t "~A~%" (first tree))
4   (loop for subtree in (cdr tree) do
5     (mytab (+ depth 1))
6     (format t "- `A" (first subtree))
7     (if (atom (second subtree))
8       (format t " -> `A~%" (second subtree))
9       (progn (terpri) (print-tree (second subtree) (+ depth 5))))))
10
11 (defun mytab (n)
12   (loop for i from 1 to n do (format t " ")))

```

De forma que:

```

CL-ID3> (print-tree *)
CIELO
- SOLEADO
  HUMEDAD
  - NORMAL -> SI
  - ALTA -> NO
- NUBLADO -> SI
- LLUVIA
  VIENTO
  - FUERTE -> NO
  - DEBIL -> SI
NIL

```

Aunque en realidad lo que necesitamos es una interfaz gráfica.

6.9. Interfaz gráfica para cl-id3

Una vez que definimos la dependencia entre los archivos de nuestro sistema vía ASDF y el paquete para nuestra aplicación vía `defpackage`, podemos pensar en definir una interfaz gráfica para `cl-id3`. Lo primero es incluir un archivo `cl-id3-gui.lisp` en la definición del sistema. El paquete `cl-id3` ya hace uso de `capl`, la librería para interfaces gráficas de Lispworks. La interfaz finalizada se muestra en la figura 6.3. En esta sección abordaremos la implementación de la interfaz.

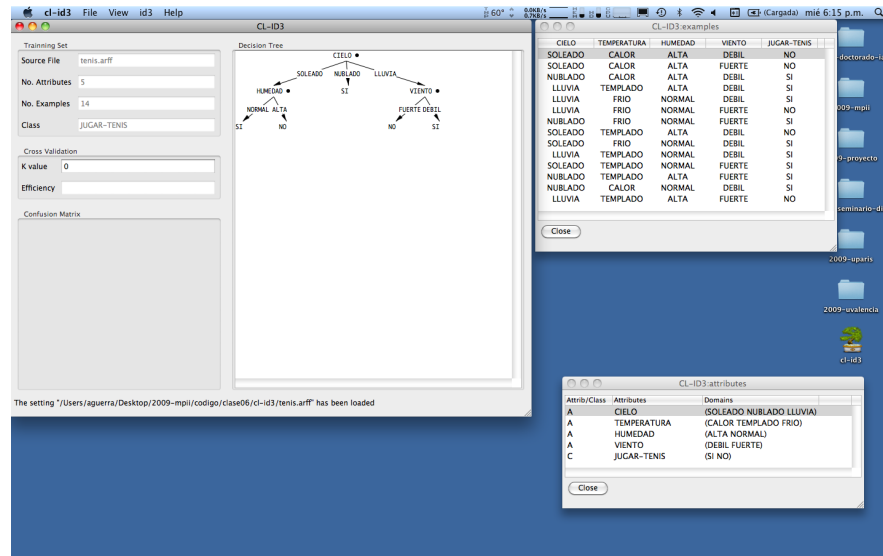


Figura 6.3 La interfaz gráfica de la aplicación `cl-id3`.

Como es usual, la interfaz incluye una barra de menús, una ventana principal para desplegar el árbol inducido e información sobre el proceso de inducción; y varias ventanas auxiliares para desplegar ejemplos, atributos y demás información adicional. En el escritorio puede verse el icono `cl-id3` (un bonsái) que permite ejecutar nuestra aplicación.

El código de la interfaz se puede dividir conceptualmente en dos partes: una que incluye la definición de los elementos gráficos en ella y otra que define el comportamiento de esos elementos en la interacción con el usuario.

6.9.1. Definiendo la interfaz

Los componentes gráficos de la interfaz y la manera en que estos se despliegan, se define haciendo uso de la función `define-interface` del `capi` como se muestra a continuación:

```

1 (define-interface cl-id3-gui ()
2   ()
3   (:panes
4     (source-id-pane text-input-pane
5       :accessor source-id-pane
6       :text ""
7       :enabled nil)
8     (num-attributes-pane text-input-pane
9       :accessor num-attributes-pane
10      :text ""
11      :enabled nil)
12     (num-examples-pane text-input-pane
13       :accessor num-examples-pane
14       :text ""
15       :enabled nil)
16     (class-pane text-input-pane
17       :accessor class-pane
18       :text ""
19       :enabled nil)
20     (efficiency-pane text-input-pane
21       :text ""
22       :enabled nil)
23     (k-value-pane text-input-pane
24       :text "0")
25     (tree-pane graph-pane
26       :title "Decision Tree"
27       :title-position :frame
28       :children-function 'node-children
29       :edge-pane-function
30       #'(lambda (self from to)
31           (declare (ignore self from))
32           (make-instance
33             'labelled-arrow-pinboard-object
34             :data (princ-to-string (node-from-label to))))
35       :visible-min-width 450
36       :layout-function :top-down)
37     (state-pane title-pane
38       :accessor state-pane
39       :text "Welcome to CL-ID3.))
40   (:menus
41     (file-menu
42       "File"
43       ("Open" :selection-callback 'gui-load-file
44         :accelerator #\o)
45       ("Quit" :selection-callback 'gui-quit
46         :accelerator #\q))
47     (view-menu
48       "View"
49       ("Attributes" :selection-callback 'gui-view-attributes
50         :accelerator #\a
51         :enabled-function #'(lambda (menu) *attributes-on*))
52       ("Examples" :selection-callback 'gui-view-examples
53         :accelerator #\e
54         :enabled-function #'(lambda (menu) *examples-on*)))
55     (id3-menu
56       "id3"
57       ("Induce" :selection-callback 'gui-induce
58         :accelerator #\i

```

```

59         :enabled-function #'(lambda (menu) *induce-on*)
60     ("Classify" :selection-callback 'gui-classify
61              :accelerator #\k
62              :enabled-function #'(lambda (menu) *classify-on*))
63     ("Cross-validation" :selection-callback 'gui-cross-validation
64                       :accelerator #\c
65                       :enabled-function
66                       #'(lambda (menu) *cross-validation-on*)))))
67
68     (help-menu
69      "Help"
70      ("About" :selection-callback 'gui-about)))
71 (:menu-bar file-menu view-menu id3-menu help-menu)
72 (:layouts
73  (main-layout column-layout '(panes state-pane))
74  (panes row-layout '(info-pane tree-pane))
75  (matrix-pane row-layout '(confusion
76                        :title "Confusion Matrix" :x-gap '10 :y-gap '30
77                        :title-position :frame :visible-min-width '200))
78  (setting-pane grid-layout
79                '("Source File" source-id-pane
80                  "No. Attributes" num-attributes-pane
81                  "No. Examples" num-examples-pane
82                  "Class" class-pane)
83                :y-adjust :center
84                :title "Training Set"
85                :title-position :frame :columns '2)
86  (id3-pane grid-layout '(("K value" k-value-pane
87                          "Efficiency" efficiency-pane)
88                          :y-adjust :center
89                          :title "Cross Validation"
90                          :title-position :frame :columns '2))
91  (:default-initargs
92   :title "CL-ID3"
93   :visible-min-width 840
94   :visible-min-height 600))

```

Hay cuatro elementos a especificar en una interfaz: paneles (elementos de la interfaz que en otros lenguajes de programación se conocen como *widgets*), menús, barra de menús y la composición gráfica de esos elementos (*layout*). Técnicamente, una interfaz es una clase con ranuras especiales para definir estos elementos y por lo tanto, la función `define-interface` es análoga a `defclass`. Estamos en el dominio del sistema orientado a objetos de Lisp, conocido como CLOS [2, 11].

En la línea 3, inicia la definición de los paneles de la interfaz con la ranura `:panes`. Aquí se definen los elementos gráficos que se utilizarán en nuestras ventanas. Por ejemplo `source-id-pane` (línea 4) es un panel de entrada de texto que inicialmente despliega una cadena vacía y está deshabilitado (el usuario no puede escribir en él). El panel `tree-pane` (línea 25) es un panel gráfico que nos permitirá visualizar el árbol inducido. La función `node-children` se encargará de computar los hijos de la raíz del árbol, para dibujarlos. Como deseamos que los arcos entre nodos estén etiquetados con el valor del atributo padre, redefinimos el tipo de arco en la ranura `:edge-pane-function` (línea 29). Como podrán deducir de la función anónima ahí definida, necesitaremos cambiar la representación interna del árbol inducido para hacer más sencilla su visualización. La línea 37 define un panel de tipo título para implementar una barra de estado del sistema.

A partir de la línea 40 definimos los menús del sistema. Todos ellos tienen *shortcuts* definidos en las ranuras `:accelerator`. La ranura `:enabled-function`

me permite habilitar y deshabilitar los menús según convenga, con base en las siguientes variables globales:

```

1 (defvar *examples-on* nil "t enables the examples menu")
2 (defvar *attributes-on* nil "t enables the attributes menu")
3 (defvar *induce-on* nil "t enables the induce menu")
4 (defvar *classify-on* nil "t enables the classify menu")
5 (defvar *cross-validation-on* nil "t enables the cross-validation menu")

```

Si el valor de las variables cambia a `t`, el menú asociado se habilita.

El comportamiento de los menús está definido por la función asociada a la ranura `:selection-callback`. Las funciones asociadas a esta ranura se definen más adelante. La línea 69 define la barra de menús, es decir, el orden en que aparecen los menús definidos.

Ahora solo nos resta definir la disposición gráfica de todos estos elementos en la interfaz. Esto se especifica mediante la ranura `:layout` a partir de la línea 70. Usamos tres tipos de disposiciones: en columna (`column-layout`), en renglón (`row-layout`) y en rejilla (`grid-layout`). La columna y el renglón funcional como pilas de objetos, horizontales o verticales respectivamente. La rejilla nos permite acomodar objetos en varias columnas. El efecto es similar a definir un renglón de columnas.

Finalmente la ranura `default-initargs` permite especificar valores iniciales para desplegar la interfaz, por ejemplo el título y su tamaño mínimo, tanto horizontal como vertical.

6.9.2. Definiendo el comportamiento de la interfaz

En esta sección revisaremos las funciones asociadas a las ranuras `callback` de los componentes de la interfaz. Estas funciones definen el comportamiento de los componentes. La siguiente función se hace cargo de leer archivos que definen conjuntos de entrenamiento para `:cl-id3`:

```

1 (defun gui-load-file (data interface)
2   (declare (ignore data))
3   (let ((file (prompt-for-file
4             nil
5             :filter "*.arff"
6             :filters '("WEKA files" "*.arff"
7                       "Comme Separated Values" "*.csv"))))
8     (when file
9       (let* ((path (princ-to-string file))
10              (setting (car (last (split-sequence #\/ path)))))
11         (load-file path)
12         (setf (text-input-pane-text (source-id-pane interface))
13               setting)
14         (setf (text-input-pane-text (num-attributes-pane interface))
15               (princ-to-string (length *attributes*)))
16         (setf (text-input-pane-text (num-examples-pane interface))
17               (princ-to-string (length *examples*)))
18         (setf (text-input-pane-text (class-pane interface))
19               (princ-to-string *target*)))

```

```

20 (setf (title-pane-text (state-pane interface))
21       (format nil "The setting ~s has been loaded"
22               path))
23 (setf *examples-on* t *attributes-on* t *induce-on* t))))

```

Por defecto, estas funciones reciben como argumentos `data` e `interface` cuyo contenido son los datos en el objeto de la interfaz, por ejemplo el texto capturado; y la interfaz que hizo la llamada a la función. La función predefinida `prompt-for-file` abre un panel para seleccionar un archivo y regresa un *path* al archivo seleccionado. Podemos seleccionar el tipo de archivo que nos interesa entre las opciones ARFF y CSV. Posteriormente convertimos el camino al archivo en una cadena de texto con la función predefinida `princ-to-string` y extraemos el nombre del archivo. La serie de `setf` cambia el texto asociado a los objetos en la interfaz. La última asignación habilita los menús que permiten visualizar archivos y atributos, así como inducir el árbol de decisión.

La siguiente función destruye la interfaz que la llama, esta asociada a las opciones `quit` de la aplicación:

```

1 (defun gui-quit (data interface)
2   (declare (ignore data))
3   (quit-interface interface))

```

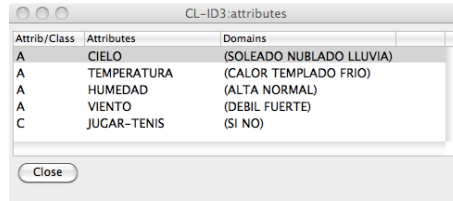
El despliegado de los atributos y sus dominios se lleva a cabo ejecutando la siguiente función:

```

1 (defun gui-view-attributes (data interface)
2   (declare (ignore data interface))
3   (let* ((max-length-attrib (apply #'max
4                                   (mapcar #'length
5                                           (mapcar #'princ-to-string
6                                                   *attributes*))))
7         (pane-total-width (list 'character
8                                 (* max-length-attrib
9                                   (+ 1 (length *attributes*)))))
10  (define-interface gui-domains () ()
11  (:panes
12   (attributes-pane multi-column-list-panel
13    :columns '(:title "Attrib/Class"
14              :adjust :left
15              :visible-min-width (character 10))
16             (:title "Attributes" :adjust :left
17              :visible-min-width (character 20))
18             (:title "Domains" :adjust :left
19              :visible-min-width (character 20)))
20    :items (loop for a in *attributes* collect
21               (list (if (eql *target* a) 'c 'a)
22                     a
23                     (get-domain a)))
24    :visible-min-width pane-total-width
25    :visible-min-height :text-height
26    :vertical-scroll t)
27   (button-pane push-button
28    :text "Close"
29    :callback 'gui-quit))
30  (:default-initargs
31   :title "CL-ID3:attributes"))
32  (display (make-instance 'gui-domains))))

```

En este caso usamos un `multi-column-list-panel` que nos permite definir columnas con encabezado. La ejecución de esta función genera una ventana como la que se muestra en la figura 6.4.



Attrib/Class	Attributes	Domains
A	CIELO	(SOLEADO NUBLADO LLUVIA)
A	TEMPERATURA	(CALOR TEMPLADO FRIO)
A	HUMEDAD	(ALTA NORMAL)
A	VIENTO	(DEBIL FUERTE)
C	JUGAR-TENIS	(SI NO)

Figura 6.4 Atributos y sus dominios desplegados en la interfaz.

La función para inducir el árbol de decisión y desplegarlo gráficamente es la siguiente:

```

1 (defun gui-induce (data interface)
2   "It induces the decision tree and displays it in the INTERFACE"
3   (declare (ignore data))
4   (setf *current-tree* (induce))
5   (display-tree (make-tree *current-tree*)
6                 interface))

```

Primero induce el árbol y después cambia su representación para poder dibujarlo. El cambio de representación hace uso de nodos que incluyen la etiqueta del arco que precede a cada nodo, excepto claro la raíz del árbol:

```

1 (defstruct node
2   (inf nil)
3   (sub-trees nil)
4   (from-label nil))
5
6 (defun make-tree (tree-as-1st)
7   "It makes a tree of nodes with TREE-AS-LST"
8   (make-node :inf (root tree-as-1st)
9             :sub-trees (make-sub-trees (children tree-as-1st))))
10
11 (defun make-sub-trees (children-1st)
12   "It makes de subtrees list of a tree with CHILDREN"
13   (loop for child in children-1st collect
14     (let ((sub-tree (second child))
15           (label (first child)))
16       (if (leaf-p sub-tree)
17           (make-node :inf sub-tree
18                     :sub-trees nil
19                     :from-label label)
20           (make-node :inf (root sub-tree)
21                       :sub-trees (make-sub-trees (children sub-tree))
22                       :from-label label))))))

```

Opcionalmente podríamos cambiar nuestro algoritmo `id3` para que generará el árbol con esta representación. Las funciones para visualizar el árbol a partir de la nueva representación son:

```

1 (defmethod print-object ((n node) stream)
2   (format stream "~s " (node-inf n)))
3
4 (defun display-tree (root interface)
5   "It displays the tree with ROOT in its pane in INTERFACE"
6   (with-slots (tree-pane) interface
7     (setf (graph-pane-roots tree-pane)
8           (list root))
9     (map-pane-children tree-pane ;; redraw panes
10      (lambda (item)
11        (update-pinboard-object item))))))
12
13 (defun node-children (node)
14   "It gets the children of NODE to be displayed"
15   (let ((children (node-sub-trees node)))
16     (when children
17       (if (leaf-p children) (list children)
18         children))))

```

La línea 9 es necesaria para acomodar los nodos una vez que el árbol ha sido dibujado totalmente, de otra forma los desplazamientos ocurridos al dibujarlo incrementalmente, pueden desajustar su presentación final. La función `node-children` es la función asociada al panel `tree-pane` para computar los hijos de un nodo. La función `print-object` especifica como queremos etiquetar los nodos del árbol.

La siguiente función nos permite desplegar la interfaz gráfica que hemos definido, después de limpiar la configuración de la herramienta:

```

1 (defun gui ()
2   (reset)
3   (display (make-instance 'cl-id3-gui)))

```

La llamada a esta función despliega la interfaz que se muestra en la figura 6.3.

Capítulo 7

Multi-procesamiento en Lisp

Resumen Normalmente, las implementaciones de Lisp pueden ejecutar diferentes hilos dentro de un solo proceso, compartiendo el mismo espacio de memoria. De hecho, Lispworks 6.0 soporta multi-procesamiento simétrico y procesos ligeros (*light-weight processes*). La ejecución de los hilos es administrada automáticamente por el sistema operativo o por el kernel de Lisp que se está usando, de forma que la tarea programada se lleva a cabo en paralelo (asíncronamente). Esta sesión trata sobre la creación y administración de hilos en Lisp, así como de la interacción entre ellos. Normalmente, y por razones históricas, en el contexto de Lisp los hilos se conocen como procesos. Desafortunadamente, el estándar de Lisp no menciona nada sobre este tema, así que lo aquí expuesto es dependiente de la implementación de LispWorks. Los ejemplos se pueden ejecutar en la versión personal de este compilador, salvo aquellos que requieren guardar una imagen nueva de Lisp, que requieren la versión profesional. Esta presentación se basa en el capítulo sobre hilos del *Common Lisp Cookbook*¹ y se debe complementar con la lectura de la sección de multi-procesamiento del manual de LispWorks 6.0.

7.1. Introducción

La primer pregunta a plantearse es ¿Porqué necesitamos preocuparnos por los hilos y los multi-procesos? En casi todos los ejemplos que hemos visto, la solución es tan directa, que no tenemos razones para ocuparnos de ellos. Pero en otros casos, es difícil imaginarse como podríamos alcanzar una solución sin multi-hilos. Estos casos incluyen, entre otros:

- Escribir un servidor capaz de responder a más de un usuario ó conexión a un tiempo dado, por ejemplo, un servidor web.
- Necesitamos que una tarea auxiliar se ejecute sin detener la tarea principal.
- Necesitamos que una tarea sea notificada de que cierto tiempo ha transcurrido.

¹ <http://cl-cookbook.sourceforge.net/process.html>

- Necesitamos mantener una tarea esperando mientras se libera algún recurso del sistema.
- Necesitamos conectarnos con un sistema que maneja multi-hilos, por ejemplo, las ventanas de una interfaz gráfica.
- Deseamos asociar diferentes contextos a diferentes partes de la aplicación, por ejemplo ligas dinámicas.
- Necesitamos hacer dos cosas al mismo tiempo.

Al comenzar a trabajar con hilos es muy fácil perder el control sobre ellos, de forma que se vuelven inactivos o comienzan a consumir grandes cantidades de recursos del CPU. Es necesario contar con un mecanismo eficiente para detener estos procesos, sin abandonar la ejecución de la imagen de Lisp. En el caso de LispWorks se puede usar el Navegador de Procesos (*Process Browser*) para detener la ejecución de los hilos. Abran un navegador (Window → Tools → Process Browser) antes de comenzar con esta práctica (los procesos mueren con el botón de la calavera, ver Figura 7.1).

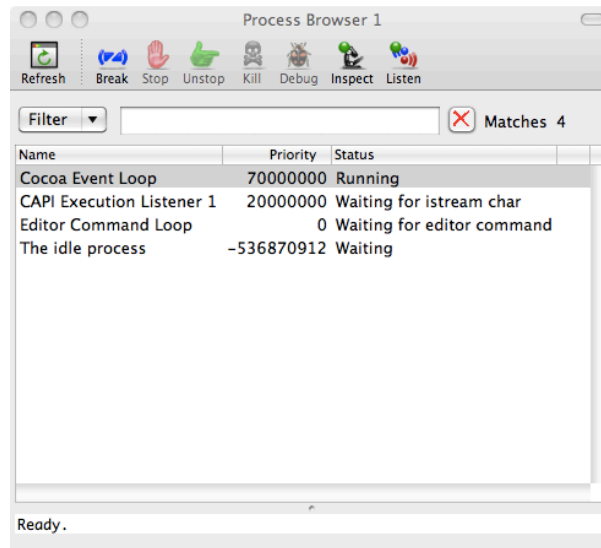


Figura 7.1 El navegador de procesos: el botón con la calavera detiene el proceso seleccionado.

Intenten crear un proceso después de abrir el navegador de procesos, evaluando la siguiente forma:

```
CL-USER 1 > (mp:process-run-function "Función en hilo" () (lambda () (loop)))
#<MP:PROCESS Name "Función en hilo" Priority 0 State "Running">
```

Un proceso con nombre `Función en hilo` debió aparecer en el navegador de procesos. Para eliminar el proceso, selecciónenlo y hagan click en el mencionado botón `kill`.

7.2. Conceptos básicos

Un proceso, en otros lenguajes llamado hilo, es un contexto de ejecución separado, con su propia pila de llamadas y ambiente dinámico. Un proceso puede estar en uno de tres estados diferentes: corriendo (*running*), en espera (*waiting*), o inactivo (*inactive*). Cuando un proceso está en espera, sigue activo, pero espera que el sistema lo despierte y le permita restaurar su ejecución. En cambio, un proceso inactivo se ha detenido por alguna razón de paro.

Para que un proceso esté activo, debe tener al menos una razón de ejecución y ninguna de paro. Si, por ejemplo, es necesario detener un proceso temporalmente, se le podría dar temporalmente una razón de paro, aunque las razones de paro no suele usarse de esta manera.

El proceso que se está ejecutando se conoce como el “proceso actual” y está identificado por el valor de la variable `mp:current-process*`. El proceso actual continua ejecutándose hasta que entra en estado de espera, invocando a la función `mp:process-wait` o a la función `mp:process-wait-with-timeout`; o permitiendo que el proceso se interrumpa a sí mismo, llamando a la función `mp:process-allow-scheduling`; o porque su tiempo disponible terminó e involuntariamente cede el control.

Bajo multi-procesamiento simétrico (Lispworks 6.X), todos los procesos que no están en espera, están corriendo y serán asignados por el sistema operativo a los CPUs disponibles. Bajo multi-procesamiento no simétrico (Lispworks 5.X y anteriores), el sistema ejecuta el proceso con la más alta prioridad. Si dos procesos tienen la misma prioridad, serán tratados de forma igualitaria y justa. A este proceso se le conoce como *scheduling Round Robin*. Esto significa que las prioridades de los procesos son majenadas de manera diferente en estas dos formas de multi-procesamiento. En el primer caso, las prioridades son prácticamente ignoradas, exceptuando que un proceso en espera con la prioridad más alta, podría despertar antes que otros procesos en espera; pero eso no está garantizado.

Para ejecutar una función en su propio hilo, es necesario hacer dos cosas:

1. Asegurar que el mecanismo multi-hilos esté siendo ejecutado. Por default, este mecanismo se ejecuta en LispWorks al usar su ambiente de desarrollo. Pero si usan una imagen que no inicia el ambiente de desarrollo, es necesario iniciar manualmente el mecanismo multi-hilos.
2. Lo que sigue es llamar a la función en su propio hilo, por ejemplo:

```
CL-USER 7 > (defvar *foo* 0)
*foo*
CL-USER 8 > (defun f () (incf *foo*))
F
CL-USER 9 > (mp:process-run-function "Incrementar *foo*" nil #'f)
#<MP:PROCESS Name "Incrementar *foo*" Priority 0 State "Dead">
CL-USER 10 > *foo*
1
```

En el ejemplo anterior, creamos un nuevo hilo llamado “Incrementar *foo*”. La función `f` fue invocada sin argumentos en ese hilo, Cuando ésta regresa, no hay nada más que hacer en el hilo así que éste termina. Observen lo siguiente:

- El primer argumento a la función `mp:process-run-function` es una cadena de caracteres que da nombre al proceso. No es necesario que los nombres sean únicos, pero es una buena práctica diferenciar sus nombres, para ayudarnos en el proceso de depuración de los programas concurrentes.
- El segundo argumento corresponde a una lista de palabras reservadas que configuran el proceso creado. Por el momento no usaremos ninguna de ellas, es decir, utilizaremos una configuración por defecto en los procesos que crearemos.
- El tercer argumento es la función que se invoca en el nuevo hilo creado. En este caso la función es `f`. El valor de este argumento puede ser cualquier símbolo que denote una función, ya sea un símbolo `fboundp` o una forma `lambda`.
- El resto de los argumentos corresponden a los parámetros de la función que se está ejecutando en el hilo. En esto, la función `mp:process-run-function` se parece a `funcall`, solo que recibe dos argumentos más al inicio.
- Esta función regresa inmediatamente un valor de tipo `mp:process`, mientras el nuevo hilo se ejecuta asincrónicamente.

7.3. Un ojo a los procesos

El sistema inicializa un cierto número de procesos al arrancar. Estos procesos están especificados por el valor de la variable `mp:*initial-processes*`. El proceso actual, como mencionamos, está especificado por el valor de la variable `mp:*current-process*`. Una lista de todos los procesos actuales es computado por la función `mp:list-all-processes`.

La función `mp:ps` es análoga a la misma función en Unix: la consola despliega los procesos corriendo en el sistema, ordenados por prioridad (y regresa `NIL`).

```
CL-USER 11 > (mp:ps)
#<MP:PROCESS Name "Cocoa Event Loop" Priority 70000000 State "Running">
#<MP:PROCESS Name "Editor Command Loop" Priority 0 State "Waiting for editor
command">
#<MP:PROCESS Name "CAPI Execution Listener 1" Priority 0 State "Running">
#<MP:PROCESS Name "The idle process" Priority -536870912 State "Waiting">
NIL
```

La función `mp:find-process-from-name` puede encontrar procesos ejecutándose en función de su nombre:

```
1 CL-USER 2 > (mp:process-run-function "sleep in the background" nil 'sleep 10)
2 #<MP:PROCESS Name "sleep in the background" Priority 0 State "Sleep">
3 CL-USER 3 > (mp:find-process-from-name "sleep in the background")
4 #<MP:PROCESS Name "sleep in the background" Priority 0 State "Sleep">
5 CL-USER 4 > (mp:find-process-from-name "sleep in the background")
6 NIL
```

De manera similar, la función `mp:process-name` regresa el nombre de un proceso. La variable `mp:*process-initial-bindings*` especifica las variables que están inicialmente acotadas en el proceso.

Cuando un proceso se ha detenido, se pueden encontrar las razones de ello con la función `mp:process-arrest-reasons`. De manera similar, la función `mp:process-run-reasons` regresa las razones por las cuales un proceso está corriendo. Ambas listas pueden cambiarse usando `setf`, pero generalmente no es necesario modificar las razones de paro. Las prioridades de los procesos pueden especificarse explícitamente al iniciar su corrida con la palabra clave `:priority`.

Veamos otro ejemplo. La siguiente función imprime una tabla de multiplicar del número `number` de uno hasta `total`.

```
1 (defun print-table (number total stream)
2   (do ((i 1 (+ i 1)))
3       ((> i total))
4     (format stream "~S x ~S = ~S~%"
5             number i (* i number))
6     (mp:process-allow-scheduling)))
```

de forma que si ejecutamos la función, tendremos:

```
CL-USER 48 > (print-table 2 10 *standard-output*)
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
NIL
```

Si queremos correr la función en un hilo, definimos:

```
1 (defun process-print-table (name number total)
2   (mp:process-run-function name
3                           nil
4                           #'print-table number total *standard-output*))
```

y ejecutamos:

```
CL-USER 49 > (process-print-table "t1" 1 10)
#<MP:PROCESS Name "t1" Priority 0 x State "1Waiting to lock buffer for
modification"> =
1
  x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
...
1 x 10 = 10
```

¿Porqué la salida en consola es un poco rara? Prueben computar dos tablas de multiplicar, por ejemplo, con ayuda de `mapcar`. Podrán observar que no es posible saber en qué orden y como se entrelazarán las ejecuciones de los dos hilos.

7.4. Cerraduras y multi-procesos

Pruben evaluar la siguiente cerradura:

```
1 (dotimes (i 10)
2   (mp:process-run-function "Una cerradura" ()
3     (lambda () (print i #.*standard-output*))))
```

Uno esperaría que la salida fuese un listado del 1 al 10, sin embargo obtenemos:

```
CL-USER 27 > (dotimes (i 10)
  (mp:process-run-function "Una cerradura" ()
    (lambda () (print i #.*standard-output*))))
1
4
5
7
9
NIL
10
10
10
10
10
```

Esto se debe a que los 10 procesos están compartiendo la variable *i* y como mencionamos, no sabemos en qué orden se ejecutaran los procesos. Es por ello que algunos reportan el valor de *i* que encontraron al ejecutarse, e.g. 1, 4, etc. Pero Los que se ejecutan luego de que `dotimes` termino, imprimen 10, el valor de *i* que encontraron. Intentemos ahora que los procesos no compartan la variable a imprimir:

```
CL-USER 32 > (dotimes (i 10)
  (mp:process-run-function "Diez ligaduras difs" ()
    (lambda (j) (print j #.*standard-output*)
      i)))
0
1
2
3
4
6NIL
7
8
9
5
```

Vamos mejorando, aunque no podemos controlar el orden en que se ejecutan los procesos. Un efecto curioso es que en el caso anterior do cuenta de 1 a 10, cuando debería contar, como en el último caso de 0 a 9.

7.5. Esperas

En todos los ejemplos anteriores, un hilo es creado para correr una función y detenerse. En las aplicaciones reales, al menos algunos hilos deberán correr es alguna modalidad de ciclo basado en eventos. Un ciclo basado en eventos es una función que espera que un evento externo ocurra. Cuando un evento es detectado, se despacha (posiblemente a otro hilo) para ser procesado y el ciclo basado en eventos vuelve a su estado de espera. Consideren el siguiente ejemplo:

```

1 (defun flush-entries-to-file (entries-symbol max-length file)
2   (loop
3     ;; Espera a tener suficientes entradas para ir a disco.
4     (mp:process-wait (format nil "Esperado por "a entr":@p" max-length)
5                       (lambda ()
6                         (>= (length (symbol-value entries-symbol)) max-length)))
7     ;; No creamos un nuevo hilo para ejecutar la tarea
8     (let ((entries (shiftf (symbol-value entries-symbol) nil)))
9       (with-open-file (ostream file
10                        :direction :output
11                        :if-exists :append
12                        :if-does-not-exist :create)
13         (format ostream "~%Flujo de entradas:")
14         (dolist (entry (reverse entries))
15           (print entry ostream))))))

```

Para probar esta función evalúen la forma `test-flush-entries-to-file` en le siguiente listado:

```

1 (defvar *test-entries* nil)
2
3 (defvar *test-file* "~/Desktop/test.txt")
4
5 (defun test-flush-entries-to-file ()
6   (let ((tester
7         (mp:process-run-function "Probando escritura entradas en archivo" ()
8                                   'flush-entries-to-file
9                                   ' *test-entries*
10                                  10
11                                  *test-file*)))
12     (dotimes (i 100)
13       (push i *test-entries*)
14       ;; Sin el retardo ocasionado por sleep, todas las 100 entradas son
15       ;; son generadas antes de que el proceso de flujos se despierte!
16       (sleep 0.1))
17     (mp:process-kill tester)))

```

Si todo va bien, deben tener un archivo de texto `test.txt` en el escritorio de su computadora, y éste debe desplegar un flujo de entradas en bloques de 10 valores en 10 valores:

```

1 Flujo de entradas:
2 0
3 1
4 2
5 3
6 4
7 5
8 6
9 7
10 8
11 9
12 Flujo de entradas:
13 10
14 11
15 ...
16 99

```

El proceso `tester` debe morir al terminar de escribir el archivo. Verifíquelo en el navegador de procesos.

7.6. Buzones

Un buzón (*mailbox*) es una estructura diseñada para facilitar la transferencia de datos entre hilos. Existen operaciones predefinidas sobre los buzones, que son seguras, esto es, diferentes hilos pueden invocar cualquier número de estas operaciones “al mismo tiempo” sin corromper la estructura del buzón.

El siguiente ejemplo usa buzones para transferir datos generados en 10 hilos a un hilo procesador. La función `mp:mailbox-send` toma como argumentos un buzón y un objeto lisp. Los objetos enviados a un buzón se guardan en una cola FIFO y son recuperados mediante llamadas a `mp:mailbox-read`. Consideren que esta función se colgará si el buzón está vacío al invocarla. Opcionalmente se le pueden pasar razones de paro y tiempos fuera.

```

1 (defun process-data (ostream)
2   (let ((mailbox))
3     (mp:process-run-function
4       "Process data" ()
5       (lambda ()
6         ;; Crear el buzón
7         (setf mailbox (mp:make-mailbox))
8         (loop
9           ;; Espera a que alguien escriba en el buzón
10          (let ((datum (mp:mailbox-read mailbox
11                    "Esperado datos a procesar."
12                    5)))
13            ;; Procesa el resultado
14            (if datum
15              (format ostream "~&Processing ~a.~%" datum)
16              ;; Termina si no hay datos
17              (return))))))
18     (mp:process-wait "Esperado a que el buzón exista."
19                    (lambda () mailbox))
20     ;; Regresa el buzón para que otros puedan compartirlo.
21     mailbox)

```

La siguiente función genera 100 datos para que cada generador los envíe al procesador de datos:

```

1 (defun generate-data (id mailbox)
2   (loop for count to 100 do
3     (let ((datum (cons id count)))
4       (sleep (random 1.0))
5       (mp:mailbox-send mailbox datum))))

```

Y el demo que pasa el buzón a varios generadores:

```

1 (defun mailbox-demo ()
2   (let ((mailbox (process-data *standard-output*)))
3     (loop for id to 10 do
4       (mp:process-run-function
5         (format nil "Generator ~d." id) ()
6         'generate-data
7         id
8         mailbox))))

```

La salida del demo es como sigue:

```

CL-USER 1 > (mailbox-demo)
NIL
Processing (9 . 0).
Processing (2 . 0).
Processing (2 . 1).
Processing (9 . 1).
Processing (2 . 2).
Processing (4 . 0).
Processing (9 . 2).
Processing (3 . 0).
...
Processing (5 . 100).

```

Los procesos creados pueden verse en el navegador de procesos (Ver Figura 7.2).

7.7. Hilos e interfaz gráfica

Cada interfase del CAPI corre en su propio hilo por default. Este hilo será usado por Lisp para acciones como desplegado e invocación de llamadas (*callbacks*). Si es necesario trabajar programáticamente sobre el CAPI, se recomienda ampliamente trabajar en el hilo apropiado.

La utilidad `capl:execute-with-interface` será de ayuda para ello. Su primer argumento es una interfase, los subsecuentes argumentos son una función y sus argumentos opcionales. Esta función será ejecutada en el hilo donde corre la interfase. Para obtener la interfase de cualquier elemento del CAPI se puede usar `capl:element-interface`, como en el siguiente ejemplo, donde la única forma de cambiar el valor de `*switchable*` es ejecutar una solicitud para ello en su hilo de ejecución.

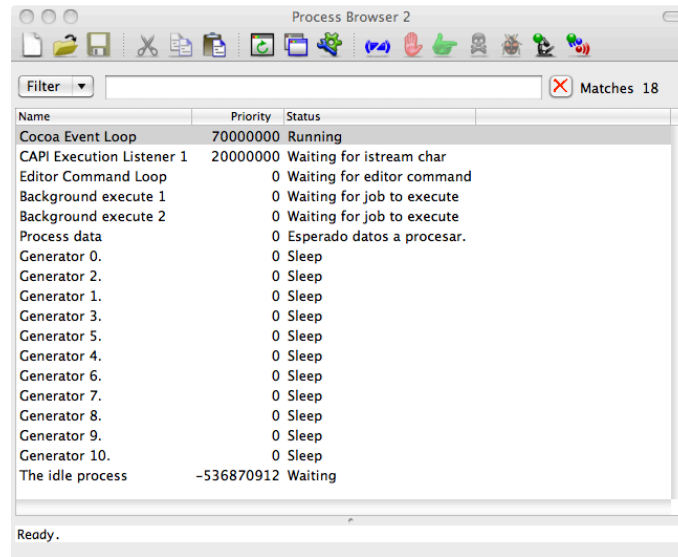


Figura 7.2 Los procesos creados por mailbox-demo. Deberan ir muriendo al concluir su tarea.

```

1 ;; Crea y despliega una ventana que puede cambiar entre sus
2 ;; dos hijos. Estos tienen diferentes colores de fondo.
3 ;; Rojo está listado antes, y por ello es visible por default.
4
5 (defvar *switchable*
6   (let ((red-pane (make-instance 'capi:output-pane
7                                 :name 'red
8                                 :background :red))
9         (green-pane (make-instance 'capi:output-pane
10                                :name 'green
11                                :background :green)))
12     (capi:contain
13       (make-instance 'capi:switchable-layout
14                      :description (list red-pane green-pane))))
15
16 ;; Utilidad para regresar el hijo verde
17 (defun green-pane (switchable)
18   (find 'green (capi:switchable-layout-switchable-children
19              switchable)
20         :key 'capi:capi-object-name))
21
22 ;; Si intenta esto (a) obtendrá un error y (b) llamando
23 ;; (right *switchable*) no ayudará - el estado de la ventana
24 ;; se rompió, es necesario crear una nueva.
25 (defun wrong (switchable)
26   (setf (capi:switchable-layout-visible-child switchable)
27         (green-pane switchable)))
28
29 (defun right (switchable)
30   (capi:execute-with-interface
31     (capi:element-interface switchable)
32     (lambda (switchable)
33       (setf (capi:switchable-layout-visible-child switchable)
34             (green-pane switchable)))
35     switchable))

```

7.8. Locks

En algunas ocasiones es importante controlar el acceso a algún recurso, de tal forma que solo un hilo pueda operar sobre él a un tiempo dado. Un método para lograr esto, consiste en definir un conjunto de hilos especializados que pueden acceder al recurso y hacer que otros hilos que quieran usar el recurso, lo hagan escribiendo en buzones si quieren que los hilos especializados accedan al recurso en su beneficio. Sin embargo hay dos potenciales problemas con este enfoque:

- Frecuentemente el hilo que invoca tiene que esperar a que la operación en el recurso se complete.
- Es una forma pesada de hacer algo que tendría que ser un proceso sencillo.

Un *lock* es un objeto que solo puede ser accedido por un hilo a la vez. Un hilo que intente utilizar este objeto cuando esta ocupado cambiará su estado a modo de espera, hasta que el objeto sea liberado. En el siguiente ejemplo el mecanismo se ilustra en su mínima expresión.

```

1  defvar *lock* (mp:make-lock)
2
3  (defun use-resource-when-free (id stream)
4    ;; Los otros hilos deben esperar aquí a que el lock
5    ;; se libere, antes de que puedan procesar el cuerpo
6    ;; de esta forma.
7    (mp:with-lock (*lock*)
8      (use-resource-anyway id stream)
9      ;; Cuando salimos de la forma, el lock es liberado
10     ;; y otros hilos pueden reclamarlo.
11     ))
12
13  (defun use-resource-anyway (id stream)
14    (format stream "~&Comenzando ~a." id)
15    (sleep 1)
16    (format stream "~&Terminando ~a." id))
17
18  (defun test (lock-p)
19    (let ((run-function (if lock-p
20                          'use-resource-when-free
21                          'use-resource-anyway)))
22      (dotimes (id 3)
23        (mp:process-run-function
24          (format nil "Hilo compitiendo ~a" id) nil
25          run-function id *standard-output*)))

```

La evaluación de la última forma debe hacer que 3 hilos aparezcan como competidores en el navegador de procesos y desplegar lo siguiente:

```

CL-USER 1 > (test *lock*)
Comenzando 0.
NIL
Terminando 0.
Comenzando 1.
Terminando 1.
Comenzando 2.
Terminando 2.

```


Parte II
Teoría

Capítulo 8

Otro mundo es posible: AL

Resumen En este capítulo presentaremos **AL** [12], un lenguaje tipificado para expresar algoritmos. Este lenguaje tiene como objetivo expresar de manera precisa problemas computacionales, por lo tanto debe ser **completo**, en el sentido de que cualquier computación intuitiva, pueda ser especificada por medios finitos. Debe poseer una sintaxis que permita la construcción de algoritmos complejos a partir de partes más simples y una semántica que defina el significado de los algoritmos: Qué se va a computar y en alguna medida, cómo. Con un lenguaje como el presentado en este capítulo es posible tender un puente entre el nivel algorítmico y la especificaciones de una máquina capaz de ejecutar tales descripciones algorítmicas. El puente es el **cálculo- λ** , una teoría de las **funciones computables** que expresa propiedades elementales de operadores y operandos, aplicaciones y el rol de las variables en la computación. Ese será el tema del siguiente capítulo.

8.1. Introducción

El enfoque adoptado por AL está orientado a expresiones, donde los algoritmos mismos son expresiones que al computarse producen **valores**. Las computaciones se definen mediante un conjunto definido de **reglas de transformación**. Estas reglas se aplican sistemáticamente hasta que ya no es posible aplicar regla alguna. El valor encontrado en la última transformación es el valor que deseábamos computar.

Debe observarse que los valores tienen una interpretación más general en este lenguaje, que en los lenguajes tradicional. Los valores no son necesariamente atómicos, como los números, sino que pueden ser agregados complejos de expresiones que no pueden ser transformados en nada más y son por tanto, en cierto sentido, constantes. Tales **expresiones constantes** pueden incluir funciones, aplicaciones de funciones y variables en contextos específicos.

Denotemos a las expresiones como e o e_i con $i \in \{0, \dots, n\}$. Podemos describir una **computación** como una secuencia:

$$e_0 \rightarrow e_1 \rightarrow \cdots \rightarrow e_i \rightarrow e_{i+1} \rightarrow \cdots \rightarrow e_n$$

donde e_0 es la expresión inicial y e_n es la expresión terminal ó el valor de e_0 . La transformación de e_i a e_{i+1} se da por la aplicación de una única regla de transformación.

Las expresiones en esta secuencia tienen una propiedad muy importante: si e_n es el valor de e_0 , entonces podemos decir que ambas expresiones significan lo mismo o que tienen la misma **semántica**, aunque sintácticamente difieran. Siendo este el caso, también podemos decir que e_1 es equivalente a e_n y lo mismo para todas las expresiones en la secuencia menores que n . Es decir, las reglas de transformación que definen las computaciones llevadas a cabo **preservan el significado**, pues obviamente reemplazan iguales por iguales. Esta idea puede ilustrarse evaluando en tres transformaciones la siguiente expresión aritmética:

Ejemplo 10 *La evaluación de una expresión aritmética preserva significado:*

$$((5 + 3) \times (8/4)) \rightarrow (8 \times (8/4)) \rightarrow (8 \times 2) \rightarrow 16$$

Es evidente que todas las sub-expresiones en esta computación son, debido a las reglas de la aritmética, reemplazadas sistemáticamente de adentro hacía afuera por números. Las expresiones en la cadena difieren sintácticamente, pero son equivalentes semánticamente, tienen el mismo significado. La **equivalencia semántica**, por otro lado, no nos dice nada sobre lo que significa una expresión por si misma, otro nivel de interpretación es necesario para ello.

En un sentido estricto, podremos hablar de figuras sintácticas y transformaciones puramente sintácticas de expresiones, en otras expresiones que consideramos semánticamente equivalentes; sin que podamos hablar del significado o semántica de las expresiones en si mismas.

Aunque el lenguaje que estamos presentando no es un lenguaje estrictamente funcional, si tiene como implementación más cercana a Lisp [16] y a su intérprete *EVAL – APPLY*. Esta es la razón para comenzar la segunda parte de este curso con esta revisión.

8.2. Sintaxis de las expresiones en AL

Las expresiones simbólicas en AL incluyen **valores constantes** tales como números (por ahora no distinguiremos entre enteros y reales), cadenas de caracteres, valores booleanos (falso y verdadero), variables y los símbolos primitivos para operadores aritméticos, lógicos y relacionales. Estas expresiones se llaman **átomos** o **términos de base** (en inglés *grounded*), puesto que no están compuestos por otras expresiones del lenguaje.

Con los átomos como base, procederemos a definir algunos constructores sintácticos o **formas sintácticas** que están compuestas de expresiones y son a su vez expresiones. Estas formas se usaran para construir expresiones más complejas subs-

tituyendo expresiones en posiciones sintácticas reservadas para expresiones. En las definiciones asumimos que e_0, e_1, \dots, e_n y e denotan expresiones válidas en el lenguaje; y que v_1, v_2, \dots, v_m y f_1, \dots, f_k denotan **variables** o identificadores. Informalmente los constructores son los siguientes:

- $(e_0 e_1 \dots e_n)$ denota la **aplicación** de una expresión e_0 a las expresiones $e_1 \dots e_n$. La expresión e_0 está en la posición del **operador** y las otras expresiones en la posición de los **operandos**. Las aplicaciones son las expresiones más importantes del lenguaje ya que son a ellas que las reglas de transformación se aplican. Observen el uso de la notación prefija con fines de uniformidad.
- $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$ es una forma sintáctica especial que denota la aplicación del predicado e_0 a las expresiones consecuente e_1 y alternativa e_2 . Su objetivo es elegir e_1 o e_2 para su posterior evaluación, dependiendo del valor del predicado e_0 (`true` o `false`).
- $\text{lambda } v_1 \dots v_n \text{ in } e_0$ denota una **abstracción** de las variables $v_1 \dots v_n$ de la expresión e_0 ; o una **función sin nombre** de n **parámetros formales** $v_1 \dots v_n$ cuya **expresión cuerpo** e_0 especifica la computación de los valores funcionales. Se dice que **lambda** **liga** las variables $v_1 \dots v_n$ en el cuerpo e_0 que determina su **alcance** en lo que se conoce como **abstracción lambda**.
- $\langle e_1 \dots e_n \rangle$ denota una **lista** n -aria ó una secuencia de expresiones en un orden particular, en el que tiene sentido hablar del primer, último e i -ésimo elemento. La lista vacía se denota por $\langle \rangle$.
- $\text{letrec } f_1 = e_1 \dots f_k = e_k \text{ in } e_0$ es la expresión más compleja del lenguaje. **letrec** precede a un conjunto de **ecuaciones** que igualan las variables f_1, \dots, f_k con las expresiones e_1, \dots, e_k recursivamente y las ligan en las expresiones e_1, \dots, e_k y también en e_0 . De hecho, esas variables pueden verse como **nombres** o identificadores de funciones, por medio de las cuales puede hacerse referencia a la parte derecha de la ecuación más adelante en la expresión **letrec**. El valor de la expresión es el valor de la **expresión meta** e_0 en la cual la ocurrencia de los identificadores f_1, \dots, f_k es remplazada recursivamente por el lado derecho de sus definiciones.
- $\text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0$ es una versión no recursiva de **letrec**.
- Ninguna otra expresión es válida en el lenguaje.

De manera concisa las **formas sintácticas** pueden escribirse como:

$$\begin{aligned}
 e =_s & \text{const} \mid \text{var} \mid \text{oper} \mid \\
 & (e_0 e_1 \dots e_n) \mid \\
 & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \\
 & \text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0 \mid \\
 & \text{lambda } v_1 \dots v_n \text{ in } e_0 \mid \\
 & \langle \rangle \mid \langle e_1 \dots e_n \rangle \mid \\
 & \text{letrec } f_1 = e_1 \dots f_n = e_n \text{ in } e_0.
 \end{aligned}$$

donde el signo $=_s$ denota equivalencia sintáctica. *const* denota el conjunto de valores constantes, *var* el de variables y *oper* el de las operaciones primitivas del lenguaje AL.

8.3. Evaluación de las expresiones en AL

Ahora estamos listos para definir la forma en que las expresiones en AL serán evaluadas. Para ello solo debemos definir reglas para computar el valor de las formas sintácticas definidas en la sección anterior. Como un primer paso, lo primero que haremos será definir estas reglas de forma independientemente de cualquier mecanismo o maquinaria para proveer una idea general de como debemos proceder en principio.

Con este propósito definiremos un **evaluador abstracto** llamado EVAL. Se le debe considerar una meta-función que mapea cada forma sintáctica en otra que representa su valor, o **significado**. En este sentido, EVAL puede considerarse la **función sintáctica** del lenguaje. Este meta-función se define mediante aplicaciones recursivas sobre todas las sub-expresiones, o algunas de ellas seleccionadas, de las formas sintácticas del lenguaje. La selección de las sub-expresiones estará determinada por una **estrategia de evaluación** que debería computar valores resultantes con un casi mínimo costo computacional.

Definiremos EVAL con una estrategia intuitiva: los operandos de una aplicación generalmente deberán evaluarse antes que el operador se aplique a ellos. Si algún caso causa problemas, se aislará y se le procesará de forma diferente. Esta estrategia es directamente implementable y conlleva una alta eficiencia en tiempo de ejecución. De hecho, esta estrategia se implemente en la mayoría de los lenguajes de programación imperativos, y en ese contexto se conoce como estrategia de **llamada por valor**.

La aplicación de EVAL a una expresión e se denota $\text{EVAL}[\![e]\!]$ y recorriendo en forma descendente las formas sintácticas tenemos:

- Las expresiones atómicas como constantes, variables y operadores primitivos evalúan a si mismos:

$$\text{EVAL}[\![\textit{atomo}]\!] = \textit{atomo}$$

- Para aplicaciones con expresiones no especificadas en las posiciones de operador y operandos:

$$\text{EVAL}[\![(e_0 e_1 \dots e_n)]\!] = \text{EVAL}[\![(\text{EVAL}[\![e_0]\!] \text{EVAL}[\![e_1]\!] \dots \text{EVAL}[\![e_n]\!])]\!]$$

- Para los **condicionales** tenemos:

$$\text{EVAL}[\text{if } e_0 \text{ then } e_1 \text{ else } e_2] = \begin{cases} \text{EVAL}[e_1] & \text{Si } \text{EVAL}[e_0] = \text{true} \\ \text{EVAL}[e_2] & \text{Si } \text{EVAL}[e_0] = \text{false} \\ \text{if } \text{EVAL}[e_0] \text{ then } e_1 \text{ else } e_2 & \text{En cualquier otro caso} \end{cases}$$

Observen que si la forma e_0 no tiene un valor booleano, la expresión de queda casi idéntica, excepto que la expresión e_0 es evaluada.

- Para las expresiones `let` tenemos:

$$\text{EVAL}[\text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0] = \text{EVAL}[e_0[v_1 \leftarrow \text{EVAL}[e_1] \dots v_n \leftarrow \text{EVAL}[e_n]]]$$

esto es, la forma evalúa al valor de e_0 en donde todas las ocurrencias de las variables ligadas por `let` han sido substituidas por sus valores respectivos, mediante $\leftarrow \text{EVAL}[e_i] \mid i \in \{1, \dots, n\}$ denotando las **substituciones**.

- Para las abstracciones anónimas que ocurren en una posición distinta a la del operador, tenemos que:

$$\text{EVAL}[\text{lambda } v_1 \dots v_n \text{ in } e_0] = \text{lambda } v_1 \dots v_n \text{ in } \text{EVAL}[e_0]$$

lo que significa que las abstracciones necesitan tener su cuerpo evaluado para determinar sus valores.

- Para las listas:

$$\begin{aligned} \text{EVAL}[\langle \rangle] &= \langle \rangle \quad \text{y} \\ \text{EVAL}[\langle e_1 \dots e_n \rangle] &= \langle \text{EVAL}[e_1] \dots \text{EVAL}[e_n] \rangle \end{aligned}$$

se computan recursivamente los valores de sus sub-expresiones preservando su estructura.

- Para las expresiones `letrec`:

$$\text{EVAL}[\text{letrec } \dots f_i = e_i \dots \text{ in } e_0] = \text{EVAL}[e_0[\dots f_i \leftarrow \text{EVAL}[e_i[\dots f_i \leftarrow \text{letrec } \dots \text{ in } f_i \dots]] \dots]]$$

esta definición aparentemente complicada, debe leerse como sigue: el valor de la expresión `letrec` completa es el valor de su término meta e_0 , computado al expandir todas las ocurrencias de los **identificadores de función** f_i con los valores e_i en la mano derecha de sus **ecuaciones definitorias**. Estos valores a su vez deben computarse substituyendo las ocurrencias de los identificadores f_i en las expresiones nuevamente, por copias de las expresiones completas en `letrec`, las cuales sin embargo, tienen su expresión meta remplazada por los mismos f_i . Las formas sintácticas especializada `letrec...f_i = e_i...in f_i` de hecho representa las expresiones e_i en su forma no evaluada. Tan pronto como los f_i desaparecen de la expresión, la expresión `letrec` misma desaparece pues ya no es necesaria para que la computación continúe.

Hemos recorrido todas las formas sintáctica y sin embargo, la definición de EVAL está lejos de ser completa. Hasta el momento sólo hemos cubierto los casos generales de las aplicaciones que tienen expresiones no específicas en las posiciones de operador y operandos. Lo que falta son los casos especiales donde las expresiones en la posición del operador son (o evalúan a) abstracciones y operaciones primitivas. Estas aplicaciones definen realmente las acciones de **transformación de expresiones**, por ejemplo, los casos estándar de **aplicaciones completas** se transforman con EVAL como sigue:

$$\begin{aligned} \text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 e_1 \dots e_n) \rrbracket = \\ \text{EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket] \dots v_n \leftarrow \text{EVAL} \llbracket e_n \rrbracket] \rrbracket \end{aligned}$$

Son evaluadas al valor de los cuerpos de abstracción e_0 en los cuales todas las ocurrencias de las variables `lambda`-ligadas (los parámetros formales de las abstracciones) son substituidos de izquierda a derecha por los valores de las expresiones operando en el orden en que ocurren en las aplicaciones.

Sin embargo, la sintaxis del lenguaje también permite desigualdades entre la **aridad** n de la expresión y el número m de argumentos. En esos casos podemos optar por una aproximación cobarde, definiendo el valor de tal aplicación como una cadena de caracteres denotando el error:

$$\begin{aligned} \text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 e_1 \dots e_m) \rrbracket \mid m > n = \\ \text{“función de aridad } n \text{ recibiendo } m \text{ argumentos”} \end{aligned}$$

La computación puede entonces detenerse y regresar esa cadena como el valor de la expresión completa. De manera alternativa se puede intentar mejores soluciones, por ejemplo:

- Si el número de argumentos m es mayor que la aridad de la abstracción n , debemos definir el valor como una nueva aplicación cuyo operador es una expresión resultado de la aplicación de la abstracción de aridad n a n argumentos y cuyos operandos son los restantes $m - n$ argumentos evaluados:

$$\begin{aligned} \text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 e_1 \dots e_m) \rrbracket \mid m > n = \\ \text{EVAL} \llbracket (\text{EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket] \dots v_n \leftarrow \text{EVAL} \llbracket e_n \rrbracket] \rrbracket) \\ \text{EVAL} \llbracket e_{n+1} \rrbracket \dots \text{EVAL} \llbracket e_m \rrbracket \rrbracket \end{aligned}$$

- Si la aridad n de la abstracción excede el número de argumentos m , en cuyo caso tenemos una **aplicación parcial**, el valor es definido como sigue:

$$\begin{aligned} \text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 e_1 \dots e_m) \rrbracket \mid m < n = \\ \text{lambda } v_{m+1} \dots v_n \text{ in EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket] \dots v_m \leftarrow \text{EVAL} \llbracket e_m \rrbracket] \rrbracket \end{aligned}$$

Aunque aún no contamos con los elementos para resolver el problema, es necesario señalar que el caso donde $m < n$ puede introducir potencialmente **conflictos entre nombres** entre las variables `lambda`-acotadas renombradas que resultan de

la abstracción y las variables libres en las expresiones argumento que están siendo substituidas bajo la abstracción `lambda`. Los casos $n = m$ y $m > n$ y la evaluación de abstracciones aisladas no están completamente libres de este problema, ya que el cuerpo de la abstracción puede incluir recursivamente otras abstracciones que pueden ser penetradas por las substituciones. El **renombrado de variables** no se considera una solución apropiada a este problema, dada la complejidad inherente al crecer el tamaño y la complejidad de los algoritmos. Por el momento, si queremos seguridad al respecto, podemos adoptar una estrategia conservadora definiendo las evaluaciones parciales como:

$$\text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 \ e_1 \dots e_m) \rrbracket \mid m < n = \\ \llbracket \text{EVAL} \llbracket e_m \rrbracket \dots \text{EVAL} \llbracket e_1 \rrbracket \text{ lambda } v_1 \dots v_n \text{ in } e_0 \rrbracket$$

expresión que tiene los argumentos evaluados y la abstracción empaquetados entre corchetes, en lo que se conoce como una **cerradura**. Este mecanismo **representa** el valor de una nueva abstracción de aridad $n - m$ sin realmente computarlo., y por tanto, evitando las substituciones bajo las abstracciones y los conflictos entre nombres que las acompañan. Las cerraduras deben tratarse como abstracciones normales, es decir, pueden ser usadas como operadores y operandos de otras aplicaciones y tomar más argumentos hasta que puedan computarse llevando a a cabo las substituciones postpuestas.

Consideraciones similares con respecto a las aridades, aunque no impliquen conflictos entre nombres, aplican en el caso de las operaciones primitivas. Para las **operaciones aritméticas** (binarias) que denotaremos con `op_arit`, y usando `num`, `num1` y `num2` para denotar números, tenemos que:

$$\text{EVAL} \llbracket (\text{op_arit } e_1 \ e_2) \rrbracket = \\ \begin{cases} \text{num} & \text{Si } \text{num}_1 = \text{EVAL} \llbracket e_1 \rrbracket \wedge \text{num}_2 = \text{EVAL} \llbracket e_2 \rrbracket \\ (\text{op_arit } \text{EVAL} \llbracket e_1 \rrbracket \ \text{EVAL} \llbracket e_2 \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

esto es, el valor de la aplicación es un número si sus dos argumentos son números o evalúan a números. El valor se obtiene aplicando el operador al valor de los dos argumentos. En cualquier otro caso, la aplicación se mantiene como tal salvo que `e1` y `e2` son remplazados por sus valores.

Para las **operaciones relacionales**, denotadas por `op_rel`, tenemos que además, `str1` y `str2` denotan cadenas de caracteres, mientras que `bool` denota un valor booleano:

$$\text{EVAL} \llbracket (\text{op_rel } e_1 \ e_2) \rrbracket = \\ \begin{cases} \text{bool} & \text{Si } \text{str}_1 = \text{EVAL} \llbracket e_1 \rrbracket \wedge \text{str}_2 = \text{EVAL} \llbracket e_2 \rrbracket \\ \text{bool} & \text{Si } \text{num}_1 = \text{EVAL} \llbracket e_1 \rrbracket \wedge \text{num}_2 = \text{EVAL} \llbracket e_2 \rrbracket \\ (\text{op_rel } \text{EVAL} \llbracket e_1 \rrbracket \ \text{EVAL} \llbracket e_2 \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

donde el valor de la aplicación es un booleano si sus argumentos son numéricos o cadenas de caracteres, en cuyo caso, el operador utiliza el orden léxico para esta-

blecer su **valor de verdad**. En cualquier otro caso la aplicación permanece intacta, salvo que e_1 y e_2 son substituidos por sus valores.

Para completar esta historia, definiremos la semántica de algunas operaciones sobre listas. Aprovecharemos el hecho de que las expresiones que componen las listas no necesariamente deben evaluarse en orden para producir listas primitivas funcionalmente aplicables. Esto es, EVAL procede sobre los operandos solo cuando se puede decidir que son listas o algo más que aplicaciones:

$$\text{EVAL} \llbracket (\text{empty } e) \rrbracket = \begin{cases} \text{true} & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle \rangle \\ \text{false} & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 \dots e_n \rangle \\ (\text{empty } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{first } e) \rrbracket = \begin{cases} e_1 & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 \dots e_n \rangle \\ (\text{first } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{rest } e) \rrbracket = \begin{cases} \langle e_2 \dots e_n \rangle & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 e_2 \dots e_n \rangle \\ (\text{rest } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{append } e_1 e_2) \rrbracket = \begin{cases} \langle e_{11} \dots e_{1n} \dots e_{21} \dots e_{2m} \rangle & \text{Si } \text{EVAL} \llbracket e_1 \rrbracket = \langle e_{11} \dots e_{1n} \rangle \wedge \\ & \text{EVAL} \llbracket e_2 \rrbracket = \langle e_{21} \dots e_{2m} \rangle \\ (\text{append } \text{EVAL} \llbracket e_1 \rrbracket \text{EVAL} \llbracket e_2 \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

Varias observaciones son importantes en este punto. Primero, la definición de EVAL incluye una **verificación dinámica de tipos**. Las aplicaciones de esas funciones son evaluadas solo si sus argumentos son tipo compatibles, en cualquier otro caso quedan intactas (exceptuando que sus argumentos son substituidos por sus valores). Segundo, la meta función EVAL define la **semántica operacional** del lenguaje AL. Nos dice no solo que significa una expresión en el lenguaje (que valor tiene); pero también cómo una persona o una máquina puede computar ese valor.

Además, observen que cuando EVAL aparece en una aplicación, se propaga al frente de sus sub-expresiones, pero el EVAL al frente de la aplicación no desaparece. Esto significa que las sub-expresiones deben ser evaluadas **antes** que la aplicación completa pueda ser evaluada. Sin embargo, puesto que las sub-expresiones son sintácticamente independientes, pueden ser evaluadas en cualquier orden, incluso simultáneamente; y sus valores siempre serán los mismos!

Los EVALs son propagados recursivamente de afuera hacia adentro hasta que las sub-expresiones encontradas sean, dada su definición, valores por si mismas y permitan que su EVAL desaparezca. Lo mismo sucede con los EVALs al frente de aplicaciones cuyos componentes son todos valores atómicos. Estas formas evaluarán a algo diferente, si aplican operadores legítimos a operandos tipo compatibles; o permanecerán intactos (de la manera definida) en cuyo caso ellos mismos son su

propio valor. Estos EVALs pueden verse como **peticiones** que **fuerzan** la **evaluación** de sus sub-expresiones. Tales peticiones se satisfacen de adentro hacia afuera desapareciendo así los EVALs.

Ejemplo 11 Consideremos un ejemplo de evaluación, usando el lenguaje AL:

$$\begin{array}{c}
 \text{EVAL}[(* (- 3 (+ 2 3)) (/ 3 6))] \\
 \downarrow \\
 \text{EVAL}[(* \text{EVAL}[(- 3 (+ 2 3))] \text{EVAL}[(/ 3 6)])] \\
 \downarrow \\
 \text{EVAL}[(* \text{EVAL}[(- 3 \text{EVAL}[(+ 2 3)])] 2)] \\
 \downarrow \\
 \text{EVAL}[(* \text{EVAL}[(- 3 5)] 2)] \\
 \downarrow \\
 \text{EVAL}[(* 2 2)] \\
 \downarrow \\
 4
 \end{array}$$

Ejemplo 12 Ahora definamos factorial:

fac = lambda n in if (> n 1) then (* n (fac(- n 1))) else 1)

y evaluemos la aplicación de factorial a 5:

$$\begin{array}{c}
 \text{EVAL}[(fac 5)] \\
 \downarrow \\
 \text{EVAL}[(\text{EVAL}[(fac)] \text{EVAL}[(5)])] \\
 \downarrow \\
 \text{EVAL}[(lambda n in if (> n 1) then (* n (fac(- n 1))) else 1) 5] \\
 \downarrow \\
 \text{if EVAL}[(> 5 1)] \text{ then EVAL}[(* 5 (fac(- 5 1)))] \text{ else EVAL}[(1)] \\
 \downarrow \\
 \text{if true then EVAL}[(* 5 (fac(- 5 1)))] \text{ else EVAL}[(1)] \\
 \downarrow \\
 \text{EVAL}[(* 5 (fac(- 5 1)))] \\
 \downarrow \\
 \text{EVAL}[(* 5 \text{EVAL}[(fac(4))])] \\
 \vdots \\
 \text{EVAL}[(* 5 (* 4 (* 3 (* 2 1))))] \\
 \vdots \\
 120
 \end{array}$$

Capítulo 9

El Cálculo- λ

Resumen El Cálculo- λ es el modelo más cercano a los **algoritmos** y su **evaluación** tal y como fueron formalizados en el capítulo anterior. Se le puede considerar como el paradigma de todos los lenguajes de programación tal y como los conocemos actualmente. Es primordialmente, una teoría de las **funciones computables** que tiene que ver con propiedades fundamentales de los **operadores**, sus **aplicaciones a operandos** y con la **construcción** sistemática de operadores más complejos (algoritmos) a partir de componentes simples. Su núcleo tiene que ver con poco más que la definición de **variables**, **alcance de variables**, y la **substitución** ordenada de variables por expresiones. Se trata de un **lenguaje cerrado** en el sentido de que su **semántica** puede definirse en base a equivalencia entre expresiones (o términos) del cálculo mismo.

9.1. Introducción

El Cálculo- λ , publicado por Alonzo Church en 1932, es uno de los modelos matemáticos que se desarrollaron como respuesta inmediata a los problemas de Hilbert, a principios del siglo XX. El problema en cuestión es si existe un método mecánico general para obtener el valor de verdad de cualquier conjetura lógica, y está íntimamente relacionado con la cuestión de lo que es **algorítmicamente computable**.

Otros modelos al respecto incluyen los trabajos de Schoenfinkel y los combinadores de Curry (una forma especial de Cálculo- λ), los números de Göedel, el sistema de producción de Post, las funciones recursivas de Kleene, los algoritmos de Markov; y el más prominente con respecto a las computaciones mecánicas que puede llevar a cabo una máquina digital, la máquina de Turing. Aunque no se tenga una idea muy clara de lo que es la computabilidad, una nueva reconfortante es que todos estos modelos son equivalentes. Lo cual nos lleva a la **tesis de Church-Turing**: los problemas intuitivamente o efectivamente computables son exactamente aquellos que pueden computarse en un máquina de Turing (y por lo tanto, en los demás modelos también).

El Cálculo- λ es el modelo más cercano a los **algoritmos** y su **evaluación** tal y como fueron formalizados en el capítulo anterior. Se le puede considerar como el paradigma de todos los lenguajes de programación tal y como los conocemos actualmente. Es primordialmente, una teoría de las **funciones computables** que tiene que ver con propiedades fundamentales de los **operadores**, sus **aplicaciones a operandos** y con la **construcción** sistemática de operadores más complejos (algoritmos) a partir de componentes simples. Su núcleo tiene que ver con poco más que la definición de **variables**, **alcance de variables**, y la **substitución** ordenada de variables por expresiones. Se trata de un **lenguaje cerrado** en el sentido de que su **semántica** puede definirse en base a equivalencia entre expresiones (o términos) del cálculo mismo.

9.2. Notación del Cálculo- λ

Una función f de n variables v_1, \dots, v_n se denotan en el Cálculo- λ como:

$$f = \lambda v_1 \dots v_n. e_0$$

Sustituyendo λ por `lambda` e introduciendo `in` en lugar del punto, obtendríamos la notación empleada en las abstracciones de AL en el capítulo anterior. El símbolo f en el lado izquierdo de la ecuación denota el **nombre** o identificador de la función y puede ser referenciado en cualquier parte. La expresión del lado derecho de la ecuación define una **abstracción** de las variables v_1, \dots, v_n en el cuerpo de la función e_0 . Una definición precisa de lo que **ocurrencia libre de variables** significa, deberemos posponerla para más adelante. Por ahora basta comentar que una variable u es **libre** en la expresión e_0 si ésta no contiene ninguna abstracción de u (la variable no está en la lista $\lambda \dots$).

Una **aplicación** llamada f sobre r expresiones argumento e_1, \dots, e_r tiene la forma:

$$(f e_1 \dots e_r) = (\lambda v_1 \dots v_n. e_0 e_1 \dots e_r)$$

donde r no es necesariamente igual a n .

El caso especial de la aplicación de una abstracción a las variables abstraídas, nos da como resultado el cuerpo de la abstracción:

$$(\lambda v_1 \dots v_n. e_0 v_1 \dots v_n) = e_0$$

Para mantener el aparato formal simple y conciso, el Cálculo- λ considera, sin pérdida de generalidad, solamente abstracciones de una variable. Esto se debe al descubrimiento de Schoenfinkel y Curry que permite representar abstracciones n -arias, como n -folds anidados de abstracciones unarias (¿recuerdan el extraño nombre de **funciones Currificadas**? Pudo haber sido peor), es decir:

$$f = \lambda v_1 \dots v_n. e_0 \equiv \lambda v_1. \lambda v_2. \dots \lambda v_n. e_0$$

Usando la notación currificada, la aplicación de f a r operandos toma la forma de un r -fold de aplicaciones anidadas:

$$(f e_1 \dots e_r) \equiv (\dots((f e_1) e_2) \dots e_r)$$

Lo anterior reduce la construcción de expresiones (o términos) del Cálculo- λ a la siguiente regla sintáctica:

$$e =_s v \mid c \mid (e_0 e_1) \mid \lambda v.e_0$$

Una expresión- λ es una variable, denotada por v ; o una constante, denotada por c ; o la aplicación de una expresión e_0 a una expresión e_1 ; o la abstracción de una variable v de una expresión e_0 , respectivamente. Las expresiones que se forman a partir de la aplicación sistemática de estas reglas se conocen como **fórmulas bien formadas** (fbf) del Cálculo- λ . Cualquier otra expresión no es válida en el Cálculo- λ .

Una aplicación $(e_0 e_1)$ representa el resultado de aplicar e_0 a e_1 . Se dice que e_0 es la expresión en la **posición del operador**; y que e_1 es la expresión en la **posición del operando** de la aplicación. Es común que e_0 y e_1 se identifiquen como la **función** y el **argumento** de la aplicación, lo cual no es totalmente correcto, puesto que la sintaxis del Cálculo- λ permite que cualquier expresión válida esté en cualquiera de las dos posiciones y sólo las abstracciones y las operaciones primitivas $+$, $-$, $*$, \dots son funciones verdaderas.

Estamos interesados particularmente en aplicaciones de la forma $(\lambda v.e_0 e_1)$ que tienen una abstracción en la posición del operador y una expresión válida en la posición del operando. Estas son las expresiones relacionadas con la historia completa sobre el papel de las variables, particularmente su alcance y sustitución, en la evaluación de expresiones algorítmicas. De hecho, en ese contexto, basta considerar únicamente el **Cálculo- λ puro**, cuyo conjunto de constantes está vacío. Sin operadores primitivos, las abstracciones son las únicas funciones en juego. A pesar de esta simplificación, tenemos un modelo formal completo que provee los fundamentos necesarios para razonar acerca de la **computabilidad algorítmica**. Si se desea, podemos incluso representar números, valores de verdad, listas, entre otras cosas, como λ -abstracciones; aunque estas abstracciones lucen extrañas, sin parecido con su habitual representación.

9.3. β -reducción y α -conversión

La belleza del Cálculo- λ puro reside en que sólo necesitamos preocuparnos por una sola regla de transformación, puesto que sólo contamos con las aplicaciones de abstracciones en tal sistema. Como vimos en el capítulo anterior, segunda sección, esta regla reemplaza tales abstracciones por el cuerpo de la abstracción con las ocurrencias libres de las variables λ -ligadas, substituidas por la expresión correspondiente con el respectivo operando (o argumento). La regla se denota como:

$$\lambda v.e_0 e_1 \rightarrow_{\beta} e_0[v \leftarrow e_1]$$

La regla se conoce como **β -reducción** o **β -contracción** y se dice que **simplifica** o **reduce** el **β -redex** $(\lambda v.e_0 e_1)$ a su **reductum** o **contractum** $e_0[v \leftarrow e_1]$.

Desafortunadamente esta regla no es tan simple como parece a primera vista. Como sabemos, existen problemas con respecto a las variables ligadas en las abstracciones y la existencia de variables libres con los mismos nombres, en cuyo caso, las **substituciones** no pueden llevarse a cabo sin una acción correctiva en una de las variables con el mismo nombre. Esto concierne al **estatus de ligadura** de las variables, que debe permanecer invariante contra las β -reducciones para garantizar la **determinación** de los resultados, independiente de la elección del nombre de las variables.

Si las substituciones se llevarán a cabo de manera *naif*, es decir, con los operandos literalmente como son, por ejemplo en:

$$(\lambda u.\lambda v.u w) \rightarrow \lambda v.w \quad \text{y} \quad (\lambda u.\lambda v.u v) \rightarrow \lambda v.v$$

obtendríamos como contractum la abstracción $\lambda v.w$ en el primer caso, y $\lambda v.v$ en el segundo caso. Es evidente que la elección de la variable en la posición del operando resulta en dos funciones totalmente diferentes. En el primer caso obtendríamos una **función constante**, que independientemente del operando al que se aplique, siempre regresa w . Sin embargo, en el segundo caso obtenemos la **función identidad** que siempre reproduce la expresión operando:

$$(\lambda v.w a) \rightarrow w \quad \text{y} \quad (\lambda v.v a) \rightarrow a$$

El problema aparece en el segundo caso donde la variable libre v es substituida de manera *naif* bajo el alcance del abstracto λv y por lo tanto deviene ligada a él parasitariamente; mientras que en el primer caso substituímos la variable w que no es afectada por el abstractor λv y por lo tanto preserva su estado de ligadura de variable libre.

Podríamos decidir aceptar estas **ligaduras parásitas**, que de hecho son causadas por **conflicto entre nombres**, como se discutió en el capítulo anterior; y posiblemente sacar ventaja de ellas. Desafortunadamente, tales ligaduras destruyen dos propiedades muy útiles e importantes del Cálculo- λ que, como veremos más adelante, garantizan un comportamiento ordenado con respecto a las estrategias de evaluación, y por tanto no deben abandonarse fácilmente.

Para prevenir los conflictos entre nombres, podemos optar por una estrategia segura y demandar que todas las variables dentro de una λ -expresión tengan nombres diferentes. Sin embargo, tal estrategia no parece realista debido a razones prácticas. Al incrementarse el número de variables en algoritmos complejos, será incrementalmente más complicado inventar nombres de variables que reflejen su uso o convención. Construir nombres únicos automáticamente, por ejemplo, por enumeración, puede ser una opción siempre y cuando los nombres nuevos recuerden a los originales de alguna manera.

Aquí exploraremos una solución que requiere de una definición precisa de lo que significa una **variable libre** y una **acotada**, así como el **alcance** de una variable. Con V denotando el conjunto de variables, definimos el **conjunto de variables libres** FV de una expresión e recorriendo las tres formas sintácticas del Cálculo- λ puro y especificando lo que son las variables libres por casos:

$$FV(e) = \begin{cases} \{v\} & \text{Si } e =_s v \in V \\ FV(e_0) \cup FV(e_1) & \text{Si } e =_s (e_0 e_1) \\ FV(e_0) \setminus \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases}$$

Esta definición recursiva nos dice que el conjunto contiene solo la variable v si v es la expresión e entera; o si está en la unión de las variables libres de un operador y su operando, si e es una aplicación; o las variables que aparecen en el cuerpo de una abstracción, pero no están ligadas en ella.

Es posible ofrecer una definición complementaria del **conjunto de variables acotadas** en la expresión e :

$$BV(e) = \begin{cases} \emptyset & \text{Si } e =_s v \in V \\ BV(e_0) \cup BV(e_1) & \text{Si } e =_s (e_0 e_1) \\ BV(e_0) \cup \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases}$$

Con la ayuda de estas definiciones podemos expresar que una variable v está libre en una expresión e , si y sólo si $v \in FV(e)$; y que una variable v está acotada en una expresión e , si y sólo si $v \in BV(e)$. En una abstracción $\lambda v.e$, llamamos al cuerpo e el **alcance** del abstractor λv , lo que significa que todas las ocurrencias libres de v en e están acotadas por λv . Por ejemplo, en la expresión:

$$(\lambda u. (\lambda v. (\lambda z. (z (v u)) v) u) w)$$

la variable w está libre, puesto que no existe abstractor para ella. La variable u está acotada en la abstracción $\lambda u.(\dots)$, pero libre en su cuerpo, que es el alcance del abstractor λu . Una misma variable puede estar libre o acotada dependiendo del alcance considerado.

Una abstracción cuyo conjunto de variables libres está vacío, se dice **cerrado** o que es un **combinador**; en otro caso se dice que la abstracción es **abierto**. De gran relevancia para la implementación de lenguajes de programación basados en el Cálculo- λ , son los llamados **super combinadores**, que son abstracciones cerradas cuyos cuerpos pueden contener recursivamente sólo expresiones cerradas (o super combinadores).

Ahora estamos listos para definir de manera precisa, como la **substitución** en la β -reducción

$$(\lambda v.e_b e_a) \rightarrow_{\beta} e_b[v \leftarrow e_a]$$

se lleva a cabo: el lado derecho de esta regla de transformación debe prescribir la sustitución de todas las ocurrencias libres de la variable v en la expresión e_b por la expresión e_a . Su definición es la siguiente:

$$e_b[v \leftarrow e_a] = \begin{cases} e_a & \text{Si } e_b =_s v \in V \\ u & \text{Si } e_b =_s u \in V \wedge v \neq_s u \\ (e_0[v \leftarrow e_a] e_1[v \leftarrow e_a]) & \text{Si } e_b =_s (e_0 e_1) \\ \lambda v.e_0 & \text{Si } e_b =_s \lambda v.e_0 \\ \lambda u.e_0[v \leftarrow e_a] & \text{Si } e_b =_s \lambda u.e_0 \quad \wedge \\ & u \notin FV(e_a) \vee v \notin FV(e_b) \\ \lambda w.e_0[u \leftarrow w][v \leftarrow e_a] & \text{Si } e_b =_s \lambda u.e_0 \quad \wedge \\ & u \in FV(e_a) \wedge v \in FV(e_b) \quad \wedge \\ & w \in V \wedge w \notin FV(e_a) \cup FV(e_b) \end{cases}$$

Los últimos tres casos son muy interesantes, prescriben que debe hacerse cuando la expresión e_b , en la que las sustituciones de las ocurrencias libres de v serán llevadas a cabo, es una abstracción. Si esta abstracción liga a v , entonces no cambia puesto que no hay ocurrencias libres de v en ella. Si liga a otra variable u , entonces v puede sustituirse de manera naif por e_a si no hay ocurrencias libres de u en e_a o si tenemos el caso trivial de que v no ocurre en e_b .

El caso complementario donde la abstracción liga a u , con v ocurriendo libremente en el cuerpo de la abstracción e_0 y u ocurriendo libre en e_a , causa conflicto entre nombres: cuando e_a fuera substituida de manera naif en e_0 , las ocurrencias libres de u en e_a sería ligadas parasitariamente por el abstractor λu y cambiaría su estado de ligadura.

Existen dos formas de salir de este dilema. Podemos cambiar la variable libre v en e_a , por decir a w , o cambiar la variable ligada u a w en la abstracción. En ambos casos w debe ser una variable nueva que no haya sido usada en el β -redex original. La solución tradicional es la última, que se define en el último caso de la definición. Es más conveniente porque renombrar variables solo concierne a las variables que ocurren en el alcance de la aplicación.

La transformación que renombra:

$$\lambda u.e_0 \rightarrow_\alpha \lambda w.e_0[e \leftarrow w]$$

se conoce como α -conversión. Su realización se basa en la aplicación de una **función de α -conversión** a la abstracción cuyas variables ligadas necesitamos cambiar:

$$(\lambda v.\lambda w.(v w) \lambda u.e_0)$$

Esta transformación procede con la aplicación de las reglas definidas anteriormente en dos pasos: primero a $\lambda w.(\lambda u.e_0 w)$ y después a $\lambda w.e_0[u \leftarrow w]$. Veamos un ejemplo.

Ejemplo 13 Una secuencia de β -reducciones:

$$\begin{array}{c}
(\lambda u.(\lambda v.(\lambda z.(z u) v) u) z) \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v) u)[u \leftarrow z] \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v)[u \leftarrow z] u[u \leftarrow z]) \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v)[u \leftarrow z] z) \\
\downarrow \\
(\lambda v.(\lambda z.(z u)[u \leftarrow z] v[u \leftarrow z]) z) \\
\downarrow \\
(\lambda v.(\lambda w.(z u)[z \leftarrow w][u \leftarrow z] v) z) \\
\downarrow \\
(\lambda v.(\lambda w.(w u)[u \leftarrow z] v) z) \\
\downarrow \\
(\lambda v.(\lambda w.(w z) v) z) \\
\downarrow \\
(\lambda w.(w z) v)[v \leftarrow z] \\
\downarrow \\
(\lambda w.(w z) w) \\
\downarrow \\
(w z)[w \leftarrow z] \\
\downarrow \\
(z z)
\end{array}$$

En el ejemplo reemplazamos la variable libre w por z para evitar un conflicto entre nombres cuando los β -radices son sistemáticamente reducidos de afuera hacia adentro. En esta secuencia, la última regla de sustitución es aplicada en la quinta expresión de abajo hacia arriba para renombrar la variable ligada z en la abstracción $\lambda z.(z u)$ como w para evitar el conflicto entre nombres con la variable z que debe ser substituida por u . Pero esto es justo lo que queríamos evitar, que una variable nueva que no estaba en la expresión original apareciera, cambiando la apariencia pero no el significado de la abstracción $\lambda z.(z u)$ a $\lambda w.(w u)$. En este caso tuvimos suerte, si sólo miramos las expresiones inicial y final, ignorando los pasos intermedios, el renombrado de variables pasa desapercibido puesto que la forma normal es una

aplicación de la variable original z a sí misma, tal que preserva su estado de ligadura como variable libre en toda la secuencia de pasos de reducción.

La historia es diferente para λ -expresiones que reducen abstracciones, como:

$$(\lambda u.(\lambda v.\lambda z.((z u) v) z) v))$$

Al ejecutar las β -reducciones, de adentro hacía afuera, encontramos dos conflictos entre nombres, el primero al substituir el argumento externo v bajo λv ; y el segundo al substituir el argumento interno z bajo λz . Si renombramos v por x y luego z por y , obtenemos la abstracción $\lambda y.((y v) z)$ como resultado. Sin embargo, si invertimos el orden en el que usamos x e y , obtendríamos la abstracción $\lambda x.((x v) z)$ que es igual a la anterior, excepto que el nombre de la variable ligada ha cambiado.

Aunque la elección del nombre para las variables ligadas es totalmente irrelevante, ejecutar muchas β -reducciones que requieren cambio de nombre en un contexto mayor debe ser confuso y alienante con respecto a la expresión original, más allá de la expresión resultante.

Para evitar este problema de renombrar variables debemos recurrir a una idea más inteligente para representar el estado de ligadura de las variables, así como para lograr que la aplicación de las β -reducciones preserve el nombre de las variables introducidas en la expresión inicial, bajo toda circunstancia.

9.4. Un esquema de indexación para variables ligadas

Al especificar una abstracción, la elección de los nombres de las variables ligadas no es importante. Su única función es relacionar a los abstractores con posiciones sintácticas en el cuerpo de la abstracción. Funcionan como receptáculos donde los argumentos necesitan ser substituidos. A esta relación se le conoce como **estructura de ligado**.

Ejemplo 14 *Las dos abstracciones que se muestran a continuación son sintácticamente equivalentes módulo α -conversión de los nombres de las variables:*

$$\lambda u.\lambda v.\lambda w.(((w v) u)(x u)) \equiv \lambda x_1.\lambda x_2.\lambda x_3.(((x_3 x_2) x_1)(x x_1))$$

En casos como el anterior, cuando aplicamos las abstracciones a los mismos argumentos, obtenemos en ambos casos el mismo resultado, puesto que dos expresiones que son sintácticamente equivalentes, también lo son semánticamente.

La posición de un abstractor en una secuencia de abstractores también identifica, en el caso de aplicaciones anidadas, el nivel de anidamiento en que el argumento será tomado. A esto se le llama **estructura de substitución**. La estructura resultante, por una parte asocia abstractores con la ocurrencia de variables en el cuerpo de la abstracción; y por la otra con las posiciones de los operandos en aplicaciones anidadas. Esto se ilustra en la Figura 9.1.

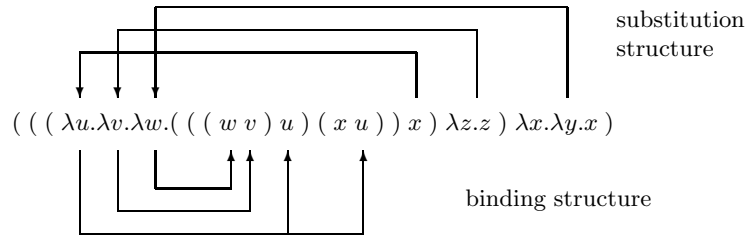


Figura 9.1 Estructuras de sustitución y ligado.

La Figura 9.2 muestra la secuencia de β -reducciones que evalúan esas aplicaciones anidadas. Necesitaremos esa secuencia para efectos de comparación más adelante. La secuencia se obtiene aplicando las β -reducciones de adentro hacia afuera: x por u , $\lambda z.z$ por v y $\lambda x.\lambda y.x$ por w ; y entonces se procede a evaluar el cuerpo instanciado, regresando el resultado de la aplicación $(x x)$ de la variable x libre a sí misma.

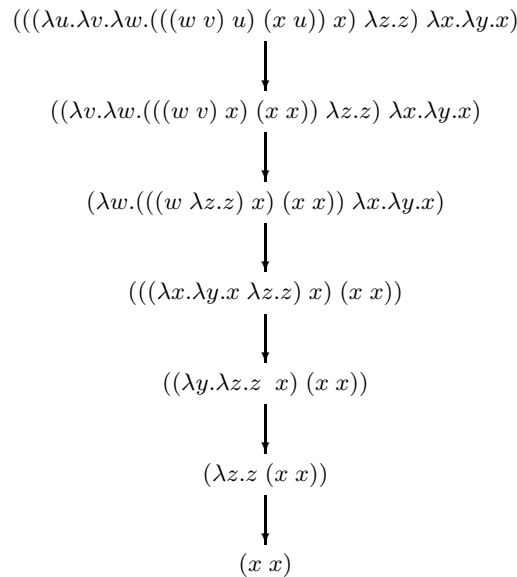


Figura 9.2 Reducción de la expresión ejemplo

Ahora procederemos a llevar el renombrado de variables un paso más adelante, llamándolas a todas x y distinguiéndolas por sus subíndices. A todas las variables se les dará el mismo nombre, por ejemplo z , y la estructura de ligado se definirá por

medio de los llamados **operadores de desligue** ó **llaves protectoras** que se colocan frente a la ocurrencia de las variables en el cuerpo de la abstracción. Estas llaves protectoras complementan a los abstractores, en un sentido amplio, deshacen ligaduras: Si una ocurrencia de la variable z está precedida por n de estos operadores, denotados como:

$$\underbrace{// \dots //}_n z$$

entonces la ocurrencia está protegida contra las n imbricaciones más internas del abstractor λz que liga a la variable. Usando esta notación, la abstracción del ejemplo puede representarse como sigue:

$$\lambda u. \lambda v. \lambda w. ((w v) u)(x u) =_s \lambda z. \lambda z. \lambda z. ((z /z) //z) (x //z)$$

Todas las variables **ligadas** se llaman ahora z y, por ejemplo, las ocurrencias de z que remplazan a la variable original u , están precedidas por dos llaves de protección $//$, de forma que los dos abstractores λz más internos, no le afecten. La variable x no se modifica porque aparece **libre** en la expresión.

El problema con las llaves de protección es que necesitan ser modificadas dinámicamente, conforme se aplican las β -reducciones. Cada que un abstractor desaparece de la abstracción original, una llave de protección debe desaparecer también. Pero también, si una variable libre entra en el alcance de un abstractor, y esta variable tiene el mismo nombre que la variable abstraída, entonces en concordancia, debemos introducir una llave de protección más. Ejemplificaremos esto aplicando nuestra abstracción ejemplo a tres abstracciones cuyas variables también se llaman z . Para hacer el ejemplo más interesante agregaremos dos abstractores al frente de la abstracción, de forma que el más externo liga ahora a lo que era la variable libre x (que ahora aparece como z con cuatro llaves protectoras, que se corresponden con los tres abstractores que ya existían y el abstractor interno de los dos que agregamos). La β -reducción de esta aplicación se muestra en la Figura 9.3. Esta reducción se lleva a cabo en el mismo número de pasos que la anterior, y produce las mismas expresiones intermedias, solo que con diferente representación.

La primera de las β -reducciones en la secuencia, incluye todo lo que debemos saber acerca del procesamiento de las llaves protectoras durante la ejecución de las reducciones. El β -redex bajo consideración está subrayado en la Figura 9.3 y su argumento es $/z$. Al hacer esto, elimina el primer λz y substituye $/z$ por todas las ocurrencias de variables ligadas en el cuerpo de la abstracción (las dos ocurrencias de $//z$). Ahora, al substituir $/z$ bajo el alcance de los dos abstractores restantes, debemos añadirle dos llaves protectoras, esto es, las dos ocurrencias de $//z$ deben ser substituidas por $///z$. para mantener la distancia correcta con el λz más externo. Pero aún hay más, la ocurrencia de la variable $///z$ que está dentro de la abstracción se verá afectada por el λz que ha desaparecido, de forma que se le debe quitar una llave protectora, lo que da lugar a tres ocurrencias de $///z$ ligadas al abstractor más externo.

Ahora podemos ser más específicos sobre la manipulación de las llaves protectoras y para ello debemos definir el estado de ligadura de las variables que preceden.

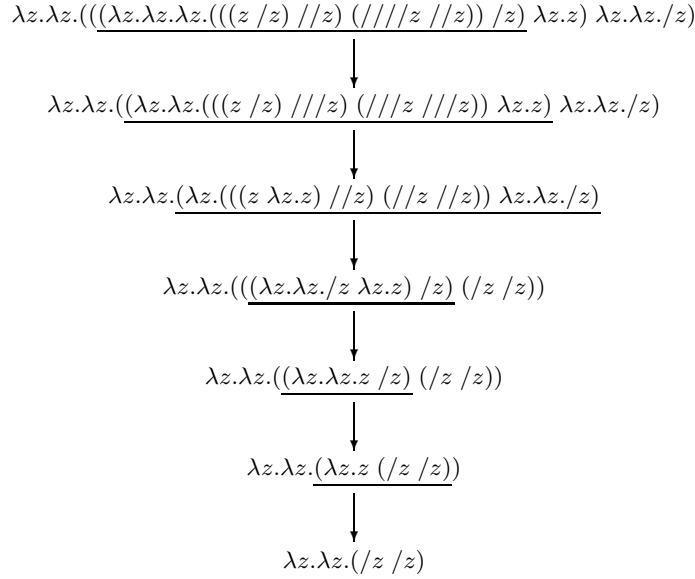


Figura 9.3 Reducción con variables de nombre único y llaves protectoras.

Sea

$$\underbrace{/ \dots //}_i v =_s / (i) \mid i \in \{0, 1, \dots\}$$

la ocurrencia de una variable v con i llaves de protección dentro de la λ -expresión e , y sea que $j \in N_o$ enumere, de la ocurrencia de la variable hacia afuera y comenzando en $j = 0$, los abstractores $\lambda v.$ que la rodean. Entonces, en relación con el abstractor j -ésimo, esta ocurrencia de la variable se dice:

$$\rightarrow \text{libre Si } i > j, \quad \rightarrow \text{acotada Si } i = j, \quad \rightarrow \text{sombreada Si } i < j$$

El índice de protección i de la ocurrencia de una variable, denota lo que se conoce como **índice de ligado** ó **distancia de ligado** (con respecto al abstractor). Con estos atributos de las ligaduras, podemos definir informalmente la regla de β -reducción en estos términos:

Dado un redex $(\lambda v. e_b e_a)$, su β -reducción regresa una expresión e'_b que se obtiene de e_b y una ocurrencia de $/^{(i)}v$ en:

- El cuerpo de la abstracción e_b :
 - Decrementar el número i de llaves de protección, si aún quedan disponibles,
 - Se substituye por e_a , si la variable está acotada,
 - No se cambia nada, si la variable está sombreada.
- El operando e_a :

- Incrementar el número i de llaves de protección en un valor k si la variable es libre o acotada con respecto al λv más interno que rodea la aplicación ($\lambda v.e_b e_a$); y si la variable penetra el alcance de k abstractores anidados λv cuando es substituida en e_b ,
- Permanece intacta si la variable está sombreada.

Se puede transformar una expresión con variables acotadas de distinto nombre, en una que tenga solo variables, digamos z , aplicando la función de α -conversión $\lambda v.\lambda z.(v z)$ a todas las abstracciones de una variable. En el caso del ejemplo, la función se inserta en la expresión original:

$$\lambda v.\lambda z.(v z)\lambda u.\dots$$

Aplicando estas α -conversiones de adentro hacía afuera y usando la regla de β -reducción definida informalmente antes, se obtiene la secuencia de reducciones que se muestra en la Figura 9.4. Esta secuencia termina con una abstracción en la que todas las ocurrencias de las variables ligadas se substituyeron por z y el número adecuado de llaves de protección ha sido introducido. La variable x no cambia por que no hay abstractor asociado a ella en toda la expresión.

$$\begin{array}{c}
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.(\lambda v.\lambda z.(v z) \lambda w.(((w v) u) (x u)))))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.\lambda z.(\lambda w.(((w v) u) (x u)) z))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.\lambda z.(((z v) u) (x u)))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.\lambda z.(\lambda v.\lambda z.(((z v) u) (x u)) z)) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.\lambda z.\lambda z.(((z/z) u) (x u))) \\
 \downarrow \\
 \lambda z.(\lambda u.\lambda z.\lambda z.(((z/z) u) (x u)) z) \\
 \downarrow \\
 \lambda z.\lambda z.\lambda z.(((z/z) //z) (x //z))
 \end{array}$$

Figura 9.4 Reducción a nombres únicos y llaves de protección adecuadas.

Evidentemente, si seguimos en esta línea de procesamiento de variables acotadas y su estructura de ligadura, el siguiente paso es un Cálculo- λ sin nombres de variables ([12], pp. 63–68) donde los abstractores están asociados a índices y la

semántica de la β -reducción se basa en la manipulación de esos índices, tal y como lo definimos aquí informalmente. Por cuestiones de tiempo, se deja al lector revisar el material asociado al Cálculo- λ sin nombres. La ventaja de este enfoque sobre el Cálculo- λ con variables nombradas es que, aunque para los humanos la reducción de las expresiones sea menos clara, su reducción automática es mucho más sencilla de implementar.

9.5. Secuencias de reducción

Revisemos algunas de las propiedades de las secuencias de β -reducciones. Dadas dos λ -expresiones e y e' , se dice que e es **β -reducible** a e' , denotado por $e \mapsto_{\beta} e'$, si y sólo si e puede ser transformado en e' por una secuencia finita (posiblemente vacía) de β -reducciones y α -conversiones. Esto produce una secuencia e_0, \dots, e_n de λ -expresiones con $e_0 =_s e$ y $e_n =_s e'$ tal que para todos los índices $i \in \{0, \dots, n-1\}$ tenemos que $e_i \rightarrow_{\beta} e_{i+1}$ ó $e_i \rightarrow_{\alpha} e_{i+1}$.

Con base en tales secuencias podemos definir que dos λ -expresiones son **semánticamente equivalentes**, denotado por $e = e'$, si y sólo si e puede ser transformada en e' por una secuencia finita (posiblemente vacía) de β -reducciones, β -reducciones reversas y α -conversiones. Esto es, para todos los índices $i \in \{0, \dots, n-1\}$, debemos tener $e_i \rightarrow_{\beta} e_{i+1}$ ó $e_{i+1} \rightarrow_{\beta} e_i$ ó $e_i \rightarrow_{\alpha} e_{i+1}$.

El objetivo de reducir una λ -expresión e es transformarla en alguna expresión e^{NF} que no contiene más redices, ó a la que no se le puedan aplicar más reglas de β -reducción. Esta expresión se conoce como la **forma normal** ó el valor de la expresión e . Si para llegar de e a e^{NF} necesitamos una secuencia finita de β -reducciones, entonces e^{NF} es también la forma normal de las λ -expresiones intermedias.

Una λ -expresión compleja que contiene diversos redices, generalmente lleva a la elección entre diferentes secuencias alternativas de β -reducciones. El problema con estas elecciones es que, iniciando de una expresión inicial, debemos alcanzar una forma normal

- para eventualmente todas las posibles secuencias, es decir, tras muchas β -reducciones finitas ejecutadas en cualquier orden; si tenemos suerte;
- para ninguna de las secuencias posibles, porque no existe una forma normal. Por ejemplo, ninguna de las secuencias termina en un número finito de pasos;
- para algunas de las secuencias posibles, pero no para todas. Esto es, algunas secuencias terminan de manera finita, pero otras no.

El último caso es muy interesante porque demanda una **estrategia** que asegure que las reducciones serán aplicadas en un orden que lleve a una forma normal, si es que ésta existe.

Consideremos un ejemplo de la primer clase de expresión: $(\lambda u.(\lambda w.(\lambda w.u) u) w)$ donde:

- procediendo de afuera hacía adentro:

$$(\lambda u.(\lambda w.(\lambda w.u) u) w) \rightarrow_{\beta} (\lambda w.(\lambda (w./w /w)w) \rightarrow_{\beta} (\lambda w./w w) \rightarrow_{\beta} w$$

- procediendo de adentro hacía afuera:

$$(\lambda u.(\lambda w.(\lambda w.u) u) w) \rightarrow_{\beta} (\lambda u.(\lambda w.u u) w) \rightarrow_{\beta} (\lambda u.u w) \rightarrow_{\beta} w$$

- y aún si comenzásemos por el radice de en enmedio, llegaríamos a la forma normal w .

Ahora, una expresión simple que no tiene forma normal es la **auto-aplicación**:

$$(\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} (\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} \dots$$

que incesantemente se reproduce a si misma.

Esta auto-aplicación servirá para definir una expresión simple cuya reducción, dependiendo del orden en que los radices son contraídos, puede terminar o no. Observen la siguiente aplicación:

$$((\lambda w.\lambda v.u \lambda w.w) (\lambda u.(u u) \lambda u.(u u)))$$

identificamos de manera inmediata que la abstracción $\lambda w.\lambda v.u$ es una **función selector** que reproduce su primer argumento, es decir $\lambda w w$, pero elimina el segundo, que en este caso particular es la misma aplicación. De forma que una reducción de afuera hacía adentro lleva a w , pero una de afuera hacía adentro no termina.

El problema con este tipo de secuencias que no terminan aunque una forma normal exista, es que tratan de reducir sub-expresiones cuya forma normal no contribuye a llegar a la forma normal de la expresión completa. En el mejor de los casos, esto atenta contra la eficiencia al ejecutar computaciones superfluas; en el peor de los casos puede llevarnos a casos de **no terminación**. Es por ello que necesitamos imperativamente garantizar la terminación con formas normales si estas existen.

La estrategia que garantiza esto se llama **reducción en orden normal**. La idea es aplicar las abstracciones a operandos no evaluados y forzar su reducción si y sólo si hay formas normales diferentes a abstracciones, por ejemplo, variables o aplicaciones que no pueden β -reducirse, en la posición del operador de las aplicaciones.

Esta estrategia debe ser definida por una **función de transformación** τ_N que mapea λ -expresiones a λ -expresiones como sigue:

$$\tau_N(e) = \begin{cases} v & \text{Si } e =_s v \in V \\ \lambda v.\tau_N(e_b) & \text{Si } e =_s \lambda v.e_b \\ \tau'_N(e) & \text{Si } e =_s (\lambda v.e_b e_2) \\ \tau'_N(\tau_N(e_1)e_2) & \text{Si } e =_s (e_1 e_2) \text{ y } e_1 \neq_s \lambda v.e_b \end{cases}$$

donde:

$$\tau'_N(e) = \begin{cases} \tau_N(e_b[v \leftarrow e_2]) & \text{Si } e =_s (\lambda v.e_b e_2) \\ (e_1 \tau_N(e_2)) & \text{Si } e =_s (e_1 e_2) \text{ y } e_1 \neq_s \lambda v.e_b \end{cases}$$

La función τ_N define un **evaluador abstracto** para expresiones del Cálculo- λ puro, similar al evaluador EVAL del capítulo anterior. A diferencia de EVAL, τ_N solo

se propaga recursivamente a través de las expresiones operador, pero no toca los operandos hasta que el operador es procesado y no es una abstracción (el último caso en τ'_N . Si es una abstracción, entonces el operando es substituido por ocurrencias libres de la variable ligada (el primer caso de τ'_N).

La reducción en orden normal también se conoce como **de afuera a adentro y de izquierda a derecha**. Esta estrategia se conoce también como **llamada por nombre** y es usada por lenguajes de programación como Algol y Simula. Es posible definir una estrategia como la usada por EVAL, conocida como **primero operandos o llamada por valor**. Intenten definir una función τ_A que implemente la llamada por valor.

La propiedad más importante de las secuencias de β -reducciones es capturada por el conocido **Teorema de Church-Rosser**. Este teorema expresa esencialmente que independientemente del orden en que las β -reducciones son llevadas a cabo sobre una λ -expresión, existe siempre una λ -expresión en la cual dos diferentes secuencias de β -reducciones pueden volverse a encontrar. Esto se puede formalizar como sigue:

Sean e_0, e_1, e_2, e_3 λ -expresiones. Entonces $e_0 \mapsto_{\beta} e_1$ y $e_0 \mapsto_{\beta} e_2$ implican que existe una expresión e_3 tal que $e_1 \mapsto_{\beta} e_3$ y $e_2 \mapsto_{\beta} e_3$. La prueba está más allá del contenido de este curso, pero observen que si e_3 es una forma normal, entonces esta forma es única.

Además de las formas normales, que son la meta última del proceso de β -reducción en el cálculo- λ puro, hay dos variantes intermedias de forma normal. Para distinguirlas diremos que una λ -expresión es una:

- **forma normal completa**, si no contiene β -redices. El cálculo- λ que computa formas normales completas, se conoce como normalizador completo, o simplemente normalizador;
- **forma normal débil (cabeza)**, si es una abstracción de alto nivel (contiene redices en su cuerpo) ó una aplicación de alto nivel de una abstracción n -aria a un conjunto de operandos con aridad menor a n , que están en forma débil. El cálculo- λ que computa solo formas débiles se conoce como normalizador débil; y
- **forma normal de cabeza**, si es una forma especial de abstracción de alto nivel

$$\lambda u_1 \dots \lambda u_n. (\dots (u_i e_1) \dots e_m)$$

- donde $i \in \{0, \dots, n-1\}$, cuya forma no puede cambiar más hacia la izquierda de la variable u_i ya que solo es posible llevar a cabo β -reducciones en las expresiones operando e_1, \dots, e_m . El cálculo- λ que computa formas normales de cabeza se dice normalizador de cabezas.

Las tres formas están relacionadas de la siguiente manera: toda forma normal es también una forma normal de cabeza, y toda forma normal de cabeza es también una forma normal débil, pero no a la inversa, es decir, forman una jerarquía. La normalización completa y la de cabeza, requieren de normalizadores completos puesto que necesitarán substituciones y reducciones bajo los abstractores, lo cual

puede ocasionar conflictos entre nombres. Las substituciones naif, son suficientes para la normalización débil puesto que solo se permiten reducciones de alto nivel que evitan los conflictos entre nombres.

Referencias

1. H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1–2):285–297, 1998.
2. Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. Technical Report 88-002R, X3J13, 1988.
3. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. O'Reilly, Paris, France, 2000.
4. Guy Cousinneau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
5. Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
6. Paul Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall International, 1993.
7. Paul Graham. *ANSI Common Lisp*. Prentice Hall Series in Artificial Intelligence. Prentice Hall International, 1996.
8. Alejandro Guerra-Hernández, Carlos Rubén de-la Mora-Basáñez, and Miguel Angel Jiménez-Montaño. The significance of nucleotides within DNA codons: a quantitative approach. In L. Villaseñor-Pineda and A. Martínez-García, editors, *Avances en la Ciencia de la Computación: VI Encuentro Internacional de Computación ENC'05*, pages 167–169, Puebla, Pue., México, 2005. Sociedad Mexicana de Ciencias de la Computación (SMCC), Benemérita Universidad Autónoma de Puebla.
9. Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.
10. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
11. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocol*. The MIT Press, Cambridge, MA., USA, 1991.
12. W. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer-Verlag, Berlin Heidelberg New York, 2005.
13. Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
14. Michel Mauny. Functional programming using caml light. Technical report, www.caml.org, 1995.
15. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.
16. John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.
17. T.M. Mitchell. *Machine Learning*. Computer Science Series. McGraw-Hill International Editions, Singapore, 1997.
18. Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kauffman Publishers, 1992.
19. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
20. J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA., USA, 1993.
21. Didier Rémy. *Using, Understanding, and Unraveling the Ocaml Language*. Didier Rémy, 2001.
22. Peter Seibel. *Practical Common Lisp*. Apress, USA, 2005.
23. C. Shannon and W. Weaver. The mathematical theory of communication. *The Bell System Technical Journal*, 27:623–656, July, October 1948.
24. Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers, Amsterdam Boston Heidelberg London New York Oxford Paris San Diego San Francisco Sigapore Sydney Tokyo, 2005.

