Tema 2: Seminario de Scheme

Bibliografía

Este seminario está basado en los siguientes materiales. Os recomendamos que los leáis y, si os interesa y os queda tiempo, que exploréis también en los enlaces que hemos dejado en los apuntes para ampliar información.

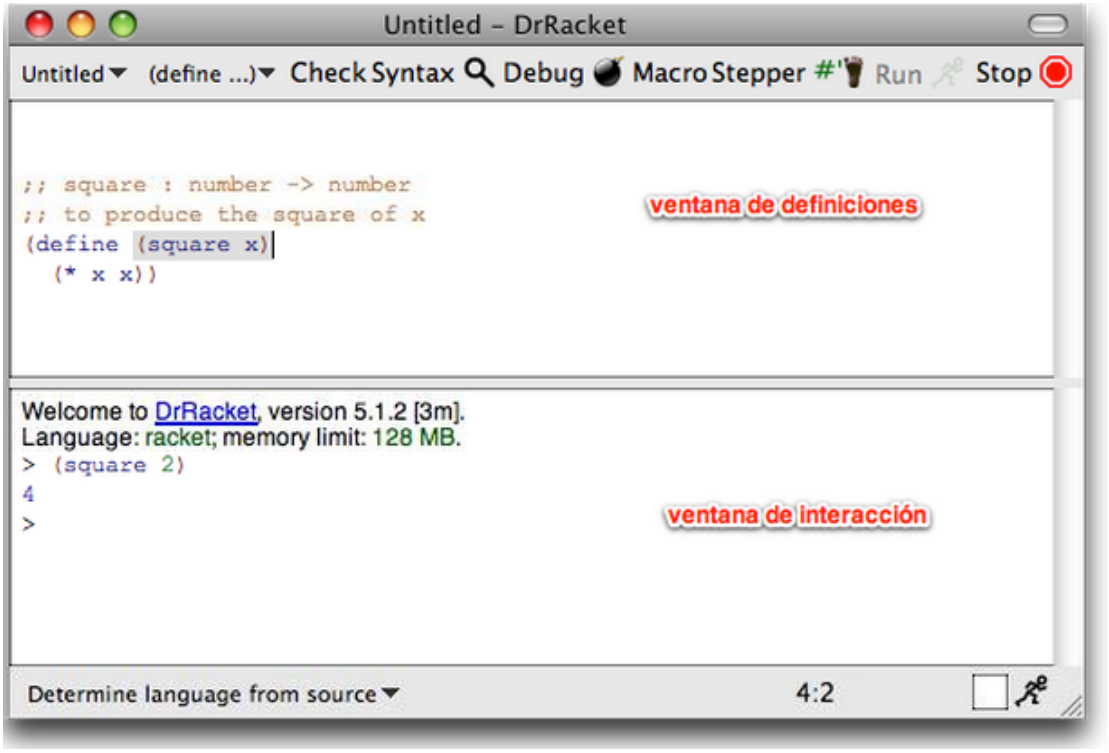
- DrRacket, Entorno de programación DrRacket
- R5RS
- Simply Scheme

El lenguaje de programación Scheme

Scheme es un lenguaje de programación que surgió en los laboratorios del MIT en 1975, cuando Guy L. Steele y Gerarld J. Sussman buscaban un lenguaje con una semántica muy clara y sencilla. Pensaban que los lenguajes no se deberían desarrollar añadiendo muchas características, sino quitando las debilidades y las limitaciones que hacen que las características adicionales parezcan necesarias. Scheme es un dialecto de Lisp, es un lenguaje interpretado, muy expresivo y soporta varios paradigmas. Estuvo influenciado por el cálculo lambda. El desarrollo de Scheme ha sido lento, ya que la gente que estandarizó Scheme es muy conservadora en cuanto a añadirle nuevas características,porque la calidad ha sido siempre más importante que la utilidad empresarial. Por eso Scheme es considerado como uno de los lenguajes mejor diseñados de propósito general. Aprender Scheme hará que seáis mejores programadores cuando utilicéis otros lenguajes de programación.

El entorno de programación DrRacket

Cuando lanzamos DrRacket, vemos que tiene tres partes: una fila de botones arriba, dos paneles de edición en el medio y una barra de estado abajo.



image

El panel de edición superior es la ventana de definiciones. Se utiliza para definir funciones, como la función square en el ejemplo. El panel inferior, llamado *ventana de interacción*, se utiliza para evaluar expresiones interactivamente. Pulsando el botón *Run*, se evalúa el programa de la *ventana de definiciones*, haciendo que esas definiciones estén disponibles en la ventana de interacción. Así, dada la definición de `square`, después de pulsar *Run*, podemos teclear la expresión `(square 2)` en la *ventana de interacción*, se evaluará y mostrará el resultado, en este caso 4.

Cambiar el idioma del entorno

Podemos interactuar con el entorno en el idioma que queramos. Si queremos tenerlo por ejemplo en español, vamos al menú *Help -> Interactúa con DrRacket en español*. Nos aparecerá un diálogo que nos obligará a reiniciar el intérprete para aceptar los cambios.

image

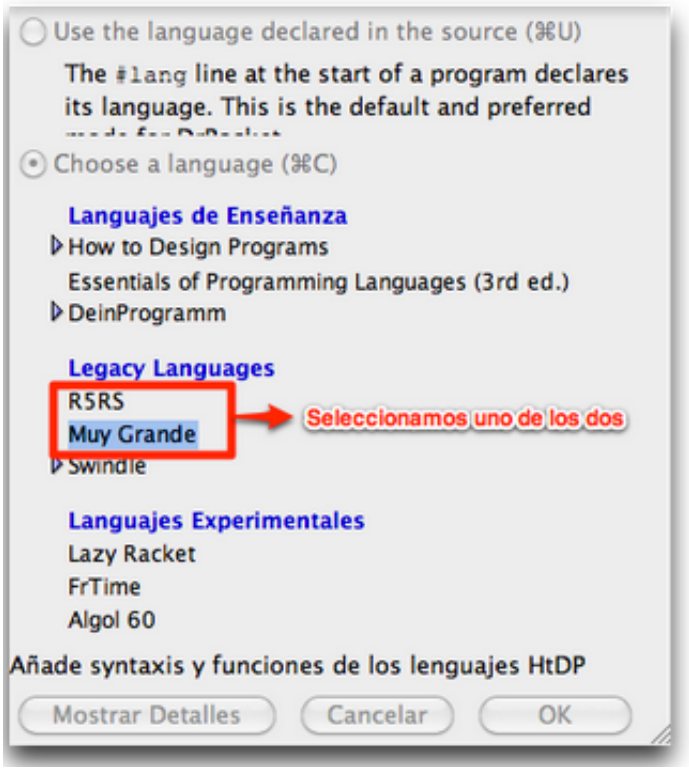
Eligiendo un lenguaje

DrRacket soporta muchos dialectos de Scheme, los cuales se pueden agrupar en 4 tipos:

- Legacy Languages
- How to Design Programs Teaching Languages
- Experimental Languages
- Output Printing Styles

A nosotros nos interesa el primer grupo, *Legacy Languages*, y dentro de ellos el *R5RS* y el *Pretty Big* (muy grande).
El lenguaje *R5RS* contiene todas las primitivas y sintaxis definidas en el estándar *R5RS*. El lenguaje *Pretty Big* proporciona un lenguaje compatible con las primeras versiones de Scheme e importa el módulo *mzscheme*, entre otros.

Nota: En el tema de programación funcional podríamos utilizar ambos lenguajes, aunque vamos a optar por el *R5RS*, y cuando lleguemos al tema de programación imperativa cambiaremos al lenguaje Muy Grande, porque necesitaremos la librería *mzscheme* para trabajar con parejas mutables.
Para seleccionar un lenguaje pinchamos en el menú: *Lenguaje -> Seleccionar Lenguaje*.



image

Seleccionamos *R5RS* dentro de *Legacy Languages*, aceptamos y a continuación pulsamos el botón *Run* (Ejecutar) para que se cargue ese lenguaje. La próxima vez que arranquemos DrRacket nuestros cambios se habrán guardado y no tendremos que volver a seleccionar un lenguaje.

El lenguaje Scheme

Vamos a empezar probando algunos ejemplos

Scheme es un lenguaje interpretado. Vamos a lanzar DrRacket y teclear en la ventana de

interacción algunas expresiones. El intérprete analizará la expresión y mostrará el valor resultante de evaluarla.

```
2
(+ 2 3)
(+)
(+ 2 4 5 6)
(+ (* 2 3) (- 3 1))
```

Las expresiones en Scheme tienen una forma denominada *notación prefija de Cambridge* (el nombre de Cambridge es por la localidad Cambridge, Massachusets, donde reside el MIT, lugar en el que se ideó el Lisp), en la que la expresión está delimitada por paréntesis y el operador va seguido de los operandos. La sintaxis es la siguiente:

```
(<función> <arg1> ... <argn>)
```

En Scheme podemos interpretar los paréntesis abiertos '(' como evaluadores o lanzadores de la función que hay a continuación. La forma de evaluar una expresión en Scheme es muy sencilla:

1. Evaluar cada uno de los argumentos
2. Aplicar la función nombrada tras el paréntesis a los valores resultantes de la evaluación anterior

```
(+ (* 2 3) (- 3 (/ 12 3)))
⇒ (+ 6 (- 3 (/ 12 3)))
⇒ (+ 6 (- 3 4))
⇒ (+ 6 -1)
⇒ 5
```

En Scheme los términos función y procedimiento significan lo mismo y se usan de forma intercambiable. Son ejemplos de funciones o procedimientos: +, -, *. En Scheme la evaluación de una función siempre devuelve un valor, a no ser que se produzca un error que detiene la evaluación:

```
(* (+ 3 4) (/ 3 0))
```

Definiendo variables y funciones

Podemos utilizar en el intérprete la forma especial `define` para definir variables y funciones. En clase de teoría veremos cómo es el funcionamiento del `define`, pero por el momento lo utilizaremos para definir variables asociadas a valores, y funciones o procedimientos. Scheme es un lenguaje multiparadigma pero principalmente funcional, y una de sus características principales es que los programas se construyen mediante la definición de funciones.

Definimos variables en la ventana de interacción:

```
(define pi 3.14159)
pi
(sin (/ pi 2))
(define a (+ 2 (* 3 4)))
a
```

Para definir una función también se utiliza define, con la siguiente sintaxis:

```
(define (<nombre-funcion> <args>)
    <cuerpo-funcion>
)
```

Por ejemplo, vamos a definir una función que toma dos números como parámetros y devuelve la suma de sus cuadrados:

```
(define (suma-cuadrados x y)
    (+ (* x x) (* y y)))
(suma-cuadrados 2 3) ⇒ 13
```

Podemos comprobar una característica muy importante de Scheme. Se trata de un lenguaje débilmente tipeado, en el que los argumentos `x` e `y` no tienen tipo. Si se invoca a la función pasando algún dato que no sea un número, el intérprete no detectará ningún error hasta que el momento en que se intente evaluar la multiplicación. Lo podemos comprobar con el siguiente ejemplo:

```
(suma-cuadrados 10 "hola")
```

Veremos más adelante que hay distintos tipos de números, y la función definida va a funcionar bien para todos ellos. En el ejemplo anterior hemos pasado como parámetro números enteros. Podemos pasar números reales:

```
(suma-cuadrados 2.4 5.8) ⇒ 39.4
```

O fracciones:

```
(suma-cuadrados (/ 2 3) (/ 3 5)) ⇒ 181/225
```

Algunas primitivas

Las primitivas de Scheme consisten en un conjunto de tipos de datos, formas especiales y funciones incluidas en el lenguaje. A lo largo del curso iremos introduciendo estas primitivas. Las primitivas del lenguaje están descritas en las 21 páginas del apartado 6 (Standard procedures) del R5RS.

Vamos a revisar los tipos de datos primitivos de Scheme, así como algunas funciones primitivas para trabajar con valores de esos tipos.

- Booleanos
- Números
- Caracteres
- Cadenas
- Parejas
- Listas

Estos dos últimos los veremos en detalle en futuras clases, cuando hablemos de tipos de datos compuestos.

Boooleanos

Un booleano es un valor de verdad, que puede ser verdadero o falso. En Scheme, tenemos los símbolos #t y #f para expresar verdadero y falso respectivamente, pero en muchas operaciones se considera que cualquier valor distinto de #f es verdadero. Ejemplos:

```
#t
#f
(> 3 1.5)
(= 3 3.0)
(equal? 3 3.0)
(or (< 3 1.5) #t)
(and #t #t #f)
(not #f)
(not 3)
```

Números

La cantidad de tipos numéricos que soporta Scheme es grande, incluyendo enteros de diferente precisión, números racionales, complejos e inexactos.

Algunas primitivas sobre números

```
(<= 2 3 3 4 5)
(max 3 5 10 1000)
(/ 22 4)  ; Devuelve una fracción
(quotient 22 4)
(remainder 22 4)
(equal? 0.5 (/ 1 2))
(= 0.5 (/ 1 2))
(abs (* 3 -2))
(sin 2.2) ; relacionados: cos, tan, asin, acos, ata
```

Primitivas que devuelven números inexactos

```
(floor x) devuelve el entero más grande no mayor que x
(ceiling x) devuelve el entero más pequeño no menor que x
(truncate x) devuelve el entero más cercano a x cuyo valor absoluto no es mayor que
(round x) devuelve el entero más cercano a x, redondeado
(floor -4.3)    ⇒ -5.0
(floor 3.5)     ⇒ 3.0
(ceiling -4.3)  ⇒ -4.0
(ceiling 3.5)   ⇒ 4.0
(truncate -4.3) ⇒ -4.0
(truncate 3.5)  ⇒ 3.0
(round -4.3)    ⇒ -4.0
(round 3.5)     ⇒ 4.0
```

Operaciones sobre números

```
(number? 1)
(integer? 2.3)
(integer? 4.0)
(real? 1)
(positive? -4)
(negative? -4)
(zero? 0.2)
(even? 2)
(odd? 3)
```

Caracteres

Se soportan caracteres internacionales y se codifican en UTF–8.

```
#\a
#\A
#\space
#\ñ
#\á
```

Operaciones sobre caracteres

```
(char<? #\a #\b)
(char-numeric? \#1)
(char-alphabetic? #\3)
(char-whitespace? #\space)
(char-upper-case? #\A)
(char-lower-case? #\a)
(char-upcase #\ñ)
(char->integer #\space)
(integer->char 32) ;#\space
(char->integer (integer->char 5000))
```

Cadenas

Las cadenas son secuencias finitas de caracteres.

```
"hola"
"La palabra \"hola\" tiene 4 letras"
```

Constructores de cadenas

```
(make-string 5 #\o) ⇒ "ooooo"
(string #\h #\o #\l #\a) ⇒ "hola"
```

Operaciones con cadenas

```
(substring "Hola que tal" 2 4)
(string? "hola")
(string->list "hola")
(string-length "hola")
(string-ref "hola" 0)
(string-append "hola" "adios")
```

Comparadores de cadenas

```
(string=? "Hola" "hola")
(string=? "hola" "hola")
(string<? "aab" "cde")
(string>=? "www" "qqq")
```

Parejas

Elemento fundamental de Scheme. Es un tipo compuesto formado por dos elementos (no necesariamente del mismo tipo).

```
(cons 1 2)        ; cons crea una pareja
(cons 'a 3)       ; elementos diferentes
(car (cons "hola" 2))  ; elemento izquierdo
(cdr (cons "bye" 5))   ; elemento derecho
```

Scheme es un lenguaje débilmente tipeado y las variables y parejas pueden contener cualquier tipo de dato. Incluso otras parejas:

```
(define p1 (cons 1 2)) ; definimos una pareja formada por 1 y 2
(cons p1 3)            ; definimos una pareja formada por la pareja (1 . 2) y 3
(cons (cons 1 2) 3)    ; igual que la expresión anterior
```

Cuando evaluamos las expresiones anteriores en el intérprete, Scheme muestra el resultado de construir la pareja con la sintaxis:

```
(elemento izquierdo . elemento derecho)
```

Sin embargo hay veces que el trabajo de imprimir una pareja no es tan sencillo para Scheme. Si la pareja está formada a su vez por otras parejas, el intérprete puede imprimir cosas que no son muy comprensibles a primera vista.

Por ejemplo:

```
(cons 1 (cons 2 3))
⇒ (1 2 . 3)
```

Más adelante explicaremos por qué sucede esto. Por ahora, para evitar estos problemas, podemos utilizar la siguiente función `print-pareja` que imprime el contenido de una pareja con el formato correcto.

```
(define (print-pareja pareja)
    (if (pair? pareja)
        (begin
            (display "(")
            (print-dato (car pareja))
            (display " . ")
            (print-dato (cdr pareja))
            (display ")"))))

(define (print-dato dato)
    (if (pair? dato)
        (print-pareja dato)
        (display dato)))
```

Un ejemplo del funcionamiento:

```
(define p2 (cons 1 (cons 2 3)))
(print-pareja p2)
⇒ (1. (2 . 3))
```

Listas

Uno de los elementos fundamentales de Scheme, y de Lisp, son las listas. Es un tipo compuesto formado por un conjunto finito de elementos (no necesariamente del mismo tipo). Vamos a ver cómo definir, crear, recorrer y concatenar listas:

```
(list 1 2 3 4)     ;list crea una lista
'(1 2 3 4)         ; otra forma de definir la misma lista
(car '(1 2 3 4))   ; primer elemento de la lista
(cdr '(1 2 3 4))   ; resto de la lista
'()                ; lista vacía
(cdr '(1))         ; devuelve la lista vacía
```

```
(null? (cdr '(1))) ; comprueba si una lista es vacía
(cons 1 '(2 3 4 5)) ; nueva lista añadiendo un elemento a la cabeza
(cons 1 '())       ; lista con 1 elemento
(cons 1 (cons 2 '())) ; lista con 2 elementos
(list "hola" "que" "tal") ; lista de cadenas
(cons "hola" '(1 2 3 4))  ; lista de distintos tipos de datos
(list (list 1 2) 3 4 (list 5 6))   ; lista que contiene listas
(cons (list 1 2) '(3 4 5)) ; nueva lista añadiendo una lista
(append '(1) '(2 3 4) '(5 6)) ; nueva lista concatenando listas
(list (cons 1 2) (cons 3 4))  ; lista que contiene parejas
```

Un concepto relacionado con las listas es el de lista vacía `'()`. La comprobación de si una lista está vacía se hace con la función `null?`. Es el caso base de gran parte de funciones recursivas que recorren listas.

Importante: las instrucciones `car` y `cdr` de las listas no son exactamente iguales que el `car` y `cdr` de las parejas. El `car` de las parejas devuelve el elemento izquierdo y el `car` de las listas devuelve el primer elemento de la lista. El `cdr` de las parejas devuelve el elemento derecho y el `cdr` de las listas devuelve una nueva lista sin el primer elemento.

Estructuras de control

Como en cualquier lenguaje de programación, las estructuras de control en Scheme nos permiten seleccionar qué parte de una expresión evaluamos en función de la evaluación de una expresión condicional. Las estructuras de control las veremos con más detenimiento en las clases de teoría, ahora por el momento vamos a ver ejemplos de funcionamiento.
En Scheme tenemos dos tipos de estructuras de control: `if` y `cond`.

if

Realiza una evaluación condicional de las expresiones que la siguen, según el resultado de una condición.

```
(define x 5)
(if (> x 2)
    #t
    #f)
```

cond

Cuando tenemos un conjunto de alternativas o para evitar usar ifs anidados.
`cond` evalúa una serie de condiciones y devuelve el valor de la expresión asociada a la primera condición verdadera.

```
(cond
    ((> 3 4) “3 es mayor que 4”)
    ((< 2 1) “2 es menor que 1”)
```

```
    ((> 3 2) “3 es mayor que 2”)
    (else “ninguna condicion es cierta”))
```

Comentarios

Para comentar una línea de código en la ventana de definiciones, se escribe el símbolo punto y coma `;` al comienzo de la línea.
Si queremos comentar más de una línea, podemos utilizar el menú de DrRacket: seleccionamos las líneas a comentar y pinchamos en la opción Racket -> comentar con punto y coma.

Ejemplos completos

Raíz de segundo grado

Vamos a resolver la ecuación de segundo grado en Scheme. Vamos a implementar el procedimiento `(ecuacion a b c)` que devuelva una pareja con las dos raíces de la solución. Nos vamos a ayudar de funciones auxiliares.

Recordamos la fórmula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Solución:

```
(define (discriminante a b c)
    (- (* b b) (* 4 a c)))

(define (raiz-pos a b c)
    (/ (+ (* b -1) (sqrt (discriminante a b c))) (* 2 a)))

(define (raiz-neg a b c)
    (/ (- (* b -1) (sqrt (discriminante a b c))) (* 2 a)))

(define (ecuacion a b c)
    (cons (raiz-pos a b c) (raiz-neg a b c)))
```

Lo probamos:

```
(ecuacion 1 -5 6)
(3 . 2)
(ecuacion 2 -7 3)
(3 . 1/2)
(ecuacion -1 7 -10)
(2 . 5)
```

Conversión de grados Celsius a Farenheit

Vamos a definir una función llamada `convertir-temperatura` que permite realizar una conversión de grados Fahrenheit a Centígrados o vicerversa.

La función toma dos argumentos, el primero será un número que representa los grados y el segundo será un carácter (`F` o `C`) que indica la unidad de medida en la que están expresados los grados.

Las fórmulas de conversión son las siguientes:

$$C = (F - 32) * 5/9$$

$$F = (C * 9/5) + 32$$

Primero definimos unas funciones auxiliares que calculan las expresiones anteriores:

```
(define (a-grados-fahrenheit grados-centigrados)
  (+ (* (/ 9 5) grados-centigrados) 32))

(define (a-grados-centigrados grados-fahrenheit)
  (* (/ 5 9) (- grados-fahrenheit 32)))
```

Y ahora ya podemos definir la función principal:

```
(define (convertir-temperatura grados tipo)
  (cond ((equal? tipo #\F) (list (a-grados-centigrados grados) "grados centigrados"))
        ((equal? tipo #\C) (list (a-grados-fahrenheit grados) "grados fahrenheit"))
        (else "tipo de cambio incorrecto")))
```

Por ejemplo:

```
(convertir-temperatura 50 #\F) ; => (10 "grados centigrados")
(convertir-temperatura 50 #\C) ; => (122 "grados fahrenheit")
```

Ejercicios

Ejercicio 1

Lanza DrRacket y escribe cada una de las siguientes instrucciones en el intérprete. Están ordenadas por dificultad de arriba abajo y de izquierda a derecha. ¡¡Piensa en los resultados!!. Intenta entender cómo interpreta Scheme lo que escribes.

Instrucción	Instrucción
3	(+ (- 4 (* 3 (/ 4 2) 4)) 3)

(+ 1 2)	(* (+ (+ 2 3) 4) (* (* 3 3) 2))
(+ 1 2 3 4)	(* (+ 2 3 4 5) 3 (- 5 2 1))
(+)	(+ (- (+ (- (+ 2 3) 5) 1) 2) 3)
(sqrt 25)	(- (sqrt (* 5 (+ 3 2))) (+ 1 1 1 1))
(* (+ 2 3) 5)	(> (* 3 (+ 2 (+ 3 1)) (+ 1 1)) (+ (* 2 2) 3))
+	(= (* 3 2) (+ 1 (+ 2 2) 1))
`#\+`	(not (> (+ 3 2) 5))
“+”	(and (even? 2) (odd? (+ 3 2)))
“hola”	(remainder (+ 6 2) (+ 1 1))

Ejercicio 2

Predice lo que devolverá Scheme cuando escribas las siguientes expresiones. Están ordenadas por dificultad de arriba abajo y de izquierda a derecha. Después, pruébalas y comprueba si tu predicción era correcta. Si no lo era, intenta comprender por qué.

Instrucción	**Instrucción**
(equal? “hola” “hola”)	(+ (char->integer(integer->char 1200))(char->integer #\A))
(string-ref “pepe” 1)	(string-length (make-string 7 #\E))
(substring “buenos dias” 1 4)	(define a 3) (define b (+ a 1))
(= “hola” “hola”)	(+ a b (* a b))
(string-ref (substring “buenos dias 2 5) 1)	(= a b)
(define pi 3.14159)	(if (and (> a b) (< b (* a b))) b a)
pi	(cond ((= a 4) 6) ((= b 4) (+ 6 7 a)) (else 25))
“pi”	(+ 2 (if (> b a) b a))
(+ pi (+ pi pi))	(* (cond ((> a b) a) ((< a b) b) (else –1)) (+ a 1))

(+ (* pi pi) (- 2 pi pi pi pi))	((if (< a b) + -) a b)

Ejercicio 3

Ejercicios con parejas. Predice lo que hace Scheme cuando escribas las siguientes expresiones. Están ordenadas por dificultad de arriba abajo y de izquierda a derecha. Después, pruébalas y comprueba si tu predicción era correcta. Si no lo era, intenta comprender por qué.

Instrucción	**Instrucción**
(cons 1 2)	(car (cons (cons 3 4) 2))
(car (cons 1 2))	(cdr (cons (cons 3 4) 2))
(cdr (cons 1 2))	(cdr (cons 1 (cons 2 3)))
(car (car (cons (cons 1 2) 3)))	(cdr (car (cons (cons 1 2) 3)))

Ejercicio 4

Ejercicios con listas. Predice lo que hace Scheme cuando escribas las siguientes expresiones. Después, pruébalas y comprueba si tu predicción era correcta. Si no lo era, intenta comprender por qué.

Instrucción	**Instrucción**
'(1 2 3 4)	(cons 3 '(1 2 3))
(cdr '(1 2 3 4))	(cdr (cons 8 (list 1 2 3 4)))
(car '(1 2 3 4))	(car (list (list 1 2) 1 2 3 4))
'(1 (2 3) (4 (5)))	(list 1 (list 2 3) (list 4 (list 5)))
(car (cdr '(1 2 3 4)))	(cons '(1 2 3) '(4 5 6))
(cdr (cdr '(1 2 3 4)))	(car (cdr (list 1 2 3 4)))
(list 1 2 3 4)	(cdr (cdr (list 1 2 3 4)))

Ejercicio 5

Los siguientes apartados intenta hacerlos sin utilizar el intérprete de Scheme.

a) Escribe una expresión diferente pero equivalente a la siguiente expresión:

```
'(1 2 (7 8 (9 (10 (2 (3))))) 3 4)
```

b) Dada la siguiente lista, indica la expresión correcta para que Scheme devuelva 5:

```
'(1 2 3 4 5 6 7 8)
```

c) Dada la siguiente lista, indica la expresión correcta para que Scheme devuelva (8).

```
'(1 2 3 4 5 6 7 8)
```

d) Dada la siguiente lista, indica la expresión correcta para que Scheme devuelva 8.

```
'(1 2 3 4 5 6 7 8)
```

e) Dada la siguiente lista, indica la expresión correcta para que Scheme devuelva (3 4).

```
'(1 2 (7 8 (9 (10 (2 (3))))) 3 4)
```

f) Dada la siguiente lista, indica la expresión correcta para que Scheme devuelva 10.

```
'(1 2 (7 8 (9 (10 (2 (3))))) 3 4)
```

g) Dada la siguiente expresión, ¿qué devuelve Scheme?

```
(cdr (cdr ‘(1 (2 3) (4 5) 6)))
```

h) Dada la siguiente expresión, ¿qué devuelve Scheme?

```
(car (cdr (cdr ‘(1 (2 3) (4 5) 6))))
```

i) Dada la siguiente expresión, ¿qué devuelve Scheme?

```
(cdr (cdr ‘(1 (2 3) 4 5)))
```
