Tema 2: Programación funcional

1. El paradigma de Programación Funcional

1.1. Pasado y presente del paradigma funcional

1.1.1. Definición y características

En una definición muy breve y concisa la programación funcional define un **programa** de la siguiente forma:

> **Definición de programa funcional**
>
> En programación funcional un programa es un conjunto de funciones matemáticas que convierten unas entradas en unas salidas, sin ningún estado interno y ningún efecto lateral.

Hablaremos más adelante de la no existencia de estado interno (variables en las que se guardan y se modifican valores) y de la ausencia de efectos laterales. Avancemos que estas son también características de la **programación declarativa** (frente a la programación tradicional imperativa que es la que se utiliza en lenguajes como C o Java). En este sentido, la programación funcional es un tipo concreto de programación declarativa.

Las características principales del paradigma funcional son:

- Definiciones de funciones matemáticas puras, sin estado interno ni efectos laterales
- Valores inmutables
- Uso profuso de la recursión en la definición de las funciones
- Uso de listas como estructuras de datos fundamentales
- Funciones como tipos de datos primitivos: expresiones lambda y funciones de orden superior

Explicaremos estas propiedades a continuación.

1.1.2. Orígenes históricos

En los años 30, junto con la máquina de Turing, se propusieron distintos modelos computacionales equivalentes que formalizaban el concepto de *algoritmo*. Uno de estos modelos fue el denominado *Cálculo lambda* propuesto por Alonzo Church en los años 30 y basado en la evaluación de expresiones matemáticas. En este formalismo los algoritmos se expresan mediante funciones matemáticas en las que puede ser usada la recursión. Una función matemática recibe parámetros de entrada y devuelve un valor. La evaluación de la función se realiza evaluando sus expresiones matemáticas mediante la sustitución de los parámetros formales por los valores reales que se utilizan en la invocación (el denominado **modelo de sustitución** que veremos más adelante).

El cálculo lambda es un formalismo matemático, basado en operaciones abstractas. Dos décadas después, cuando los primeros computadores electrónicos estaban empezando a utilizarse en grandes empresas y en universidades, este formalismo dio origen a algo mucho más tangible y práctico: un lenguaje de alto nivel, mucho más expresivo que el ensamblador, con el que expresar operaciones y funciones **que pueden ser definidas y evaluadas en el computador**, el lenguaje de programación Lisp.

1.1.3. Historia y características del Lisp

- Lisp es el primer lenguaje de programación de alto nivel basado en el paradigma funcional.
- Creado en 1958 por John McCarthy.
- Lisp fue en su época un lenguaje revolucionario que introdujo nuevos conceptos de programación no existentes entonces: funciones como objetos primitivos, funciones de orden superior, polimorfismo, listas, recursión, símbolos, homogeneidad de datos y programas, bucle REPL (*Read-Eval-Print Loop*)
- La herencia del Lisp llega a lenguajes derivados de él (Scheme, Golden Common Lisp) y a nuevos lenguajes de paradigmas no estrictamente funcionales, como C#, Python, Ruby, Objective-C o Scala.

Lisp fue el primer lenguaje de programación interpretado, con muchas características dinámicas que se ejecutan en tiempo de ejecución (*run-time*). Entre estas características podemos destacar la gestión de la memoria (creación y destrucción **automática** de memoria reservada para datos), la detección de excepciones y errores en tiempo de ejecución o la creación en tiempo de ejecución de funciones anónimas (expresiones *lambda*). Todas estas características se ejecutan mediante un *sistema de tiempo de ejecución* (*runtime system*) presente en la ejecución de los programas. A partir del Lisp muchos otros lenguajes han usado estas características de interpretación o de sistemas de tiempo de ejecución. Por ejemplo, lenguajes como BASIC, Python, Ruby o JavaScript son lenguajes interpretados. Y lenguajes como Java o C# tienen una avanzada plataforma de tiempo de ejecución con soporte para la gestión de la memoria dinámica (*recolección de basura*, *garbage collection*) o la compilación *just in time*.

Lisp no es un lenguaje exclusivamente funcional. Lisp se diseñó con el objetivo de ser un lenguaje de alto nivel capaz de resolver problemas prácticos de Inteligencia Artificial, no con la idea de ser un lenguaje formal basado un único modelo de computación. Por ello en Lisp (y en Scheme) existen primitivas que se salen del paradigma funcional puro y permiten programar de formar imperativa (no declarativa), usando mutación de estado y pasos de ejecución.

Sin embargo, durante la primera parte de la asignatura en la que estudiaremos la programación funcional, no utilizaremos las instrucciones imperativas de Scheme sino que escribiremos código exclusivamente funcional.

1.1.4. Lenguajes de programación funcional

En los años 60 la programación funcional definida por el Lisp fue dominante en departamentos de investigación en Inteligencia Artificial (MIT por ejemplo). En los años 70, 80 y 90 se fue relegando cada vez más a los nichos académicos y de investigación; en la empresa se impusieron los lenguajes imperativos y orientados a objetos.

En la primera década del 2000 han aparecido lenguajes que evolucionan de Lisp y que resaltan sus aspectos funcionales, aunque actualizando su sintaxis. Destacamos entre ellos:

- Clojure
- Erlang

También hay una tendencia desde mediados de la década de 2000 de incluir aspectos funcionales como las *expresiones lambda* o las funciones de orden superior en lenguajes imperativos orientados a objetos, dando lugar a lenguajes *multi-paradigma*:

- Ruby
- Python
- Groovy
- Scala
- Swift

Por último, en la década del 2010 también se ha hecho popular un lenguaje **exclusivamente funcional** como Haskell. Este lenguaje, a diferencia de Scheme y de otros lenguajes multi-paradigma, no tienen ningún elemento imperativo y consigue que todas sus expresiones sean puramente funcionales.

1.1.5. Aplicaciones prácticas de la programación funcional

En la actualidad el paradigma funcional es un **paradigma de moda**, como se puede comprobar observando la cantidad de artículos, charlas y blogs en los que se habla de él, así como la cantidad de lenguajes que están aplicando sus conceptos. Por ejemplo, solo como muestra, a continuación puedes encontrar algunos enlaces a charlas y artículos interesantes publicados recientemente sobre programación funcional:

- Lupo Montero - Introducción a la programación funcional en JavaScript (Blog)
- Andrés Marzal - Por qué deberías aprender programación funcional ya mismo (Charla en YouTube)
- Mary Rose Cook - A practical introduction to functional programming (Blog)
- Ben Christensen - Functional Reactive Programming in the Netflix API (Charla en InfoQ)

El auge reciente de estos lenguajes y del paradigma funcional se debe a varios factores, entre ellos que es un paradigma que facilita:

- la programación de sistemas concurrentes, con múltiples hilos de ejecución o con múltiples computadores ejecutando procesos conectados concurrentes.
- la definición y composición de múltiples operaciones sobre *streams* de forma muy concisa y compacta, aplicable a la programación de sistemas distribuidos en Internet.
- la programación interactiva y evolutiva.

1.1.5.1. PROGRAMACIÓN DE SISTEMAS CONCURRENTES

Veremos más adelante que una de las características principales de la programación funcional es que no se usa la *mutación* (no se modifican los valores asignados a variables ni parámetros). Esta propiedad lo hace un paradigma excelente para implementar programas concurrentes, en los que existen múltiples hilos de ejecución. La programación de sistemas concurrentes es muy complicada con el paradigma imperativo tradicional, en el que la modificación del estado de una variable compartida por más de un hilo puede

provocar *condiciones de carrera* y errores difícilmente localizables y reproducibles.

Como dice Bartosz Milewski, investigador y teórico de ciencia de computación, en su respuesta en Quora a la pregunta *¿por qué a los ingenieros de software les gusta la programación funcional?*:

> **Bartosz Milewski: ¿Por qué es popular la programación funcional?**
>
> Porque es la única forma práctica de escribir programas concurrentes. Intentar escribir programas concurrentes en lenguajes imperativos, no sólo es difícil, sino que lleva a *bugs* que son muy difíciles de descubrir, reproducir y arreglar. En los lenguajes imperativos y, en particular, en los lenguajes orientados a objetos se ocultan las mutaciones y se comparten datos sin darse cuenta, por lo que son extremadamente propensos a los errores de concurrencia producidos por las condiciones de carrera.

1.1.5.2. DEFINICIÓN Y COMPOSICIÓN DE OPERACIONES SOBRE STREAMS

El paradigma funcional ha originado un estilo de programación sobre *streams* de datos, en el que se concatenan operaciones como `filter` o `map` para definir de forma sencilla procesos y transformaciones asíncronas aplicables a los elementos del *stream*. Este estilo de programación ha hecho posible nuevas ideas de programación, como la programación *reactiva*, basada en eventos, o los *futuros* o *promesas* muy utilizados en lenguajes muy populares como JavaScript para realizar peticiones asíncronas a servicios web.

Por ejemplo, en el artículo Exploring the virtues of microservices with Play and Akka se explica con detalle las ventajas del uso de lenguajes y primitivas para trabajar con sistemas asíncronos basados en eventos en servicios como Tumblr o Netflix.

Otro ejemplo es el uso de Scala en Tumblr con el que se consigue crear código que no tiene estado compartido y que es fácilmente paralelizable entre los más de 800 servidores necesarios para atender picos de más de 40.000 peticiones por segundo:

> **Uso de Scala en Tumblr**
>
> Scala promueve que no haya estado compartido. El estado mutable se evita usando sentencias en Scala. No se usan máquinas de estado de larga duración. El estado se saca de la base de datos, se usa, y se escribe de nuevo en la base de datos. La ventaja principal es que los desarrolladores no tienen que preocuparse sobre hilos o bloqueos.

1.1.5.3. PROGRAMACIÓN EVOLUTIVA

En la metodología de programación denominada *programación evolutiva* o *iterativa* los programas complejos se construyen a base de ir definiendo y probando elementos computacionales cada vez más complicados. Los lenguajes de programación funcional encajan perfectamente en esta forma de construir programas.

Como Abelson y Sussman comentan en el libro *Structure and Implementation of Computer Programs* (SICP):

> **Abelson y Sussman sobre la programación incremental**
>
> En general, los objetos computacionales pueden tener estructuras muy complejas, y sería extremadamente inconveniente tener que recordar y repetir sus detalles cada vez que queremos usarlas. En lugar de ello, se construyen programas complejos componiendo, paso a paso, objetos computacionales de creciente complejidad.
>
> El intérprete hace esta construcción paso-a-paso de los programas particularmente conveniente porque las asociaciones nombre-objeto se pueden crear de forma incremental en interacciones sucesivas. Esta característica favorece el desarrollo y prueba incremental de programas, y es en gran medida responsable del hecho de que un programa Lisp consiste normalmente de un gran número de procedimientos relativamente simples.

No hay que confundir una metodología de programación con un paradigma de programación. Una metodología de programación proporciona sugerencias sobre cómo debemos diseñar, desarrollar y mantener una aplicación que va a ser usada por usuarios finales. La programación funcional se puede usar con múltiples metodologías de programación, debido a que los programas resultantes son muy claros, expresivos y fáciles de probar.

1.2. Evaluación de expresiones y definición de funciones

En la asignatura usaremos Scheme como primer lenguaje en el que exploraremos la programación funcional.

En el seminario de Scheme que se imparte en prácticas se estudiará en más profundidad los conceptos más importantes del lenguaje: tipos de datos, operadores, estructuras de control, intérprete, etc.

1.2.1 Evaluación de expresiones

Empezamos este apartado viendo cómo se definen y evalúan expresiones Scheme. Y después veremos cómo construir nuevas funciones.

Scheme es un lenguaje que viene del Lisp. Una de sus características principales es que las expresiones se construyen utilizando paréntesis.

Ejemplos de expresiones en Scheme, junto con el resultado de su ejecución:

```
2 ; ⇒ 2
(+ 2 3) ; ⇒ 5
(+) ; ⇒ 0
(+ 2 4 5 6) ; ⇒ 17
(+ (* 2 3) (- 3 1)) ; ⇒ 8
```

En programación funcional en lugar de decir "ejecutar una expresión" se dice "**evaluar una expresión**", para reforzar la idea de que se tratan de expresiones matemáticas que **siempre devuelven uno y sólo un resultado**.

Las expresiones se definen con una notación prefija: el primer elemento después del paréntesis de apertura

es el **operador** de la expresión y el resto de elementos (hasta el paréntesis de cierre) son sus operandos.

- Por ejemplo, en la expresión `(+ 2 4 5 6)` el operador es el símbolo `+` que representa función *suma* y los operandos son los números 2, 4, 5 y 6.
- Puede haber expresiones que no tengan operandos, como el ejemplo `(+)`, cuya evaluación devuelve 0.

Una idea fundamental de Lisp y Scheme es que los paréntesis se evalúan de dentro a fuera. Por ejemplo, la expresión

```
(+ (* 2 3) (- 3 (/ 12 3)))
```

que devuelve 5, se evalúa así:

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

La evaluación de cada expresión devuelve un valor que se utiliza para continuar calculando la expresión exterior. En el caso anterior

- primero se evalúa la expresión `(* 2 3)` que devuelve 6,
- después se evalúa `(/ 12 3)` que devuelve 4,
- después se evalúa `(- 3 4)` que devuelve -1
- y por último se evalúa `(+ 6 -1)` que devuelve 5

Cuando se evalúa una expresión en el intérprete de Scheme el resultado aparece en la siguiente línea.

1.2.2. Definición de funciones

En programación funcional las funciones son similares a las funciones matemáticas: reciben parámetros y devuelven siempre un único resultado de operar con esos parámetros.

Por ejemplo, podemos definir la función `(cuadrado x)` que devuelve el cuadrado de un número que pasamos como parámetro:

```
(define (cuadrado x)
   (* x x))
```

Después del nombre de la función se declaran sus argumentos. El número de argumentos de una función se denomina **aridad de la función**. Por ejemplo, la función `cuadrado` es una función de aridad 1, o *unaria*.

Después de declarar los parámetros, se define el cuerpo de la función. Es una expresión que se evaluará con el valor que se pase como parámetro. En el caso anterior la expresión es `(* x x)` y multiplicará el parámetro por si mismo.

Hay que hacer notar que en Scheme no existe la palabra clave `return`, sino que las funciones siempre se

definen con una única expresión cuya evaluación es el resultado que se devuelve.

Una vez definida la función `cuadrado` podemos usarla de la misma forma que las funciones primitivas de Scheme:

```
(cuadrado 10) ; ⇒ 100
(cuadrado (+ 10 (cuadrado (+ 2 4)))) ; ⇒ 2116
```

La evaluación de la última expresión se hace de la siguiente forma:

```
(cuadrado (+ 10 (cuadrado (+ 2 4)))) ⇒
(cuadrado (+ 10 (cuadrado 6))) ⇒
(cuadrado (+ 10 36)) ⇒
(cuadrado 46) ⇒
2116
```

1.2.3. Definición de funciones auxiliares

Las funciones definidas se pueden utilizar a su vez para construir otras funciones.

Lo habitual en programación funcional es definir funciones muy pequeñas e ir construyendo funciones cada vez de mayor nivel usando las anteriores.

1.2.3.1. EJEMPLO: SUMA DE CUADRADOS

Por ejemplo, supongamos que tenemos que definir una función que devuelva la suma del cuadrado de dos números. Podríamos definirla escribiendo la expresión completa, pero queda una definición poco legible.

```
; Definición poco legible de la suma de cuadrados

(define (suma-cuadrados x y)
    (+ (* x x)
       (* y y)))
```

Podemos hacer una definición mucho más legible si usamos la función `cuadrado` definida anteriormente:

```
; Definición de suma de cuadrados más legible.
; Usamos la función auxiliar 'cuadrado'

(define (cuadrado x)
    (* x x))

(define (suma-cuadrados x y)
    (+ (cuadrado x)
       (cuadrado y)))
```

Esta segunda definición es mucho más expresiva. Leyendo el código queda muy claro qué es lo que queremos hacer.

1.2.3.2. EJEMPLO: TIEMPO DE IMPACTO

Veamos otro ejemplo de uso de funciones auxiliares. Supongamos que estamos programando un juego de guerra de barcos y submarinos, en el que utilizamos las coordenadas del plano para situar todos los

elementos de nuestra flota.

Supongamos que necesitamos calcular el tiempo que tarda un torpedo en llegar desde una posición `(x1, y1)` a otra `(x2, y2)`. Suponemos que la velocidad del torpedo es otro parámetro `v`.

¿Cómo calcularíamos este tiempo de impacto?

La forma menos correcta de hacerlo es definir todo el cálculo en una única expresión. Como en programación funcional las funciones deben definirse con una única expresión debemos realizar todo el cálculo en forma de expresiones anidadas, unas dentro de otras. Esto construye una función que calcula bien el resultado. El problema que tiene es que es muy difícil de leer y entender para un compañero (o para nosotros mismos, cuando pasen unos meses):

```
;
; Definición incorrecta: muy poco legible
;
; La función tiempo-impacto devuelve el tiempo que tarda
; en llegar un torpedo a la velocidad v desde la posición
; (x1, y1) a la posición (x2, y2)
;

(define (tiempo-impacto x1 y1 x2 y2 v)
   (/ (sqrt (+ (* (- x2 x1) (- x2 x1))
               (* (- y2 y1) (- y2 y1))))
    v))
```

La función anterior hace bien el cálculo pero es muy complicada de modificar y de entender.

La forma más correcta de definir la función sería usando varias funciones auxiliares. Fíjate que es muy importante también poner los nombres correctos a cada función, para entender qué hace. Scheme es un lenguaje débilmente tipado y no tenemos la ayuda de los tipos que nos dan más contexto de qué es cada parámetro y qué devuelve la función.

```
; Definición correcta, modular y legible de la función tiempo-impacto

;
; La función 'cuadrado' devuelve el cuadrado de un número
;

(define (cuadrado x)
    (* x x))

;
; La función 'distancia' devuelve la distancia entre dos
; coordenadas (x1, y1) y (x2, y2)
;

(define (distancia x1 y1 x2 y2)
    (sqrt (+ (cuadrado (- x2 x1))
             (cuadrado (- y2 y1)))))

;
; La función 'tiempo' devuelve el tiempo que
; tarda en recorrer un móvil una distancia d a un velocidad v
;
```

```
(define (tiempo distancia velocidad)
    (/ distancia velocidad))

;
; La función 'tiempo-impacto' devuelve el tiempo que tarda
; en llegar un torpedo a la velocidad v desde la posición
; (x1, y1) a la posición (x2, y2)
;

(define (tiempo-impacto x1 y1 x2 y2 velocidad)
    (tiempo (distancia x1 y1 x2 y2) velocidad))
```

En esta segunda versión definimos más funciones, pero cada una es mucho más legible. Además las funciones como `cuadrado`, `distancia` o `tiempo` las vamos a poder reutilizar para otros cálculos.

1.2.4. Funciones puras

A diferencia de lo que hemos visto en programación imperativa, en programación funcional no es posible definir funciones con estado local. Las funciones que se definen son funciones matemáticas puras, que cumplen las siguientes condiciones:

- No modifican los parámetros que se les pasa
- Devuelven un único resultado
- No tienen estado local ni el resultado depende de un estado exterior mutable

Esta última propiedad es muy importante y quiere decir que la función siempre devuelve el mismo valor cuando se le pasan los mismos parámetros.

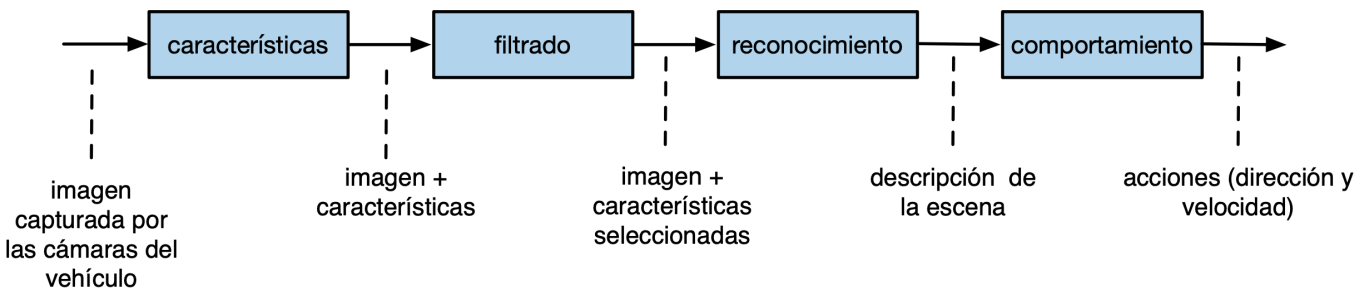
Las funciones puras son muy fáciles de entender porque no es necesario tener en cuenta ningún contexto a la hora de describir su funcionamiento. El valor devuelto únicamente depende de los parámetros de entrada.

Por ejemplo, funciones matemáticas como suma, resta, cuadrado, sin, cos, etc. cumplen esta propiedad.

1.2.5. Composición de funciones

Una idea fundamental de la programación funcional es la composición de funciones que transforman unos datos de entrada en otros de salida. Es una idea muy actual, porque es la forma en la que están planteados muchos algoritmos de procesamiento de datos en inteligencia artificial.

Por ejemplo, podemos representar de la siguiente forma el algoritmo que maneja un vehículo autónomo:



Las cajas representa funciones que transforman los datos de entrada (imágenes tomadas por las cámaras del vehículo) en los datos de salida (acciones a realizar sobre la dirección y el motor del vehículo). Las funciones intermedias representan transformaciones que se realizan sobre los datos de entrada y obtienen los datos de salida.

En un lenguaje de programación funcional como Scheme el diagrama anterior se escribiría con el siguiente código:

```
(define (conduce-vehiculo imagenes)
    (obten-acciones
        (reconoce
            (filtra
                (obten-caracteristicas imagenes)))))
```

Veremos más adelante que las expresiones en Scheme se evalúan de dentro a fuera y que tienen notación prefija. El resultado de cada función constituye la entrada de la siguiente.

En el caso de la función `conduce-vehiculo` primero se obtienen las características de las imágenes, después se filtran, después se reconoce la escena y, por último, se obtienen las acciones para conducir el vehículo.

1.3. Programación declarativa vs. imperativa

Hemos dicho que la programación funcional es un estilo de programación declarativa, frente a la programación tradicional de los lenguajes denominados imperativos. Vamos a explicar esto un poco más.

1.3.1. Programación declarativa

Empecemos con lo que conocemos todos: un **programa imperativo**. Se trata de un conjunto de instrucciones que se ejecutan una tras otra (pasos de ejecución) de forma secuencial. En la ejecución de estas instrucciones se van cambiando los valores de las variables y, dependiendo de estos valores, se modifica el flujo de control de la ejecución del programa.

Para entender el funcionamiento de un programa imperativo debemos imaginar toda la evolución del programa, los pasos que se ejecutan y cuál es el flujo de control en función de los cambios de los valores en las variables.

En la **programación declarativa**, sin embargo, utilizamos un paradigma totalmente distinto. Hablamos de *programación declarativa* para referirnos a lenguajes de programación (o sentencias de código) en los que se *declaran* los valores, objetivos o características de los elementos del programa y en cuya ejecución no existe mutación (modificación de valores de variables) ni secuencias de pasos de ejecución.

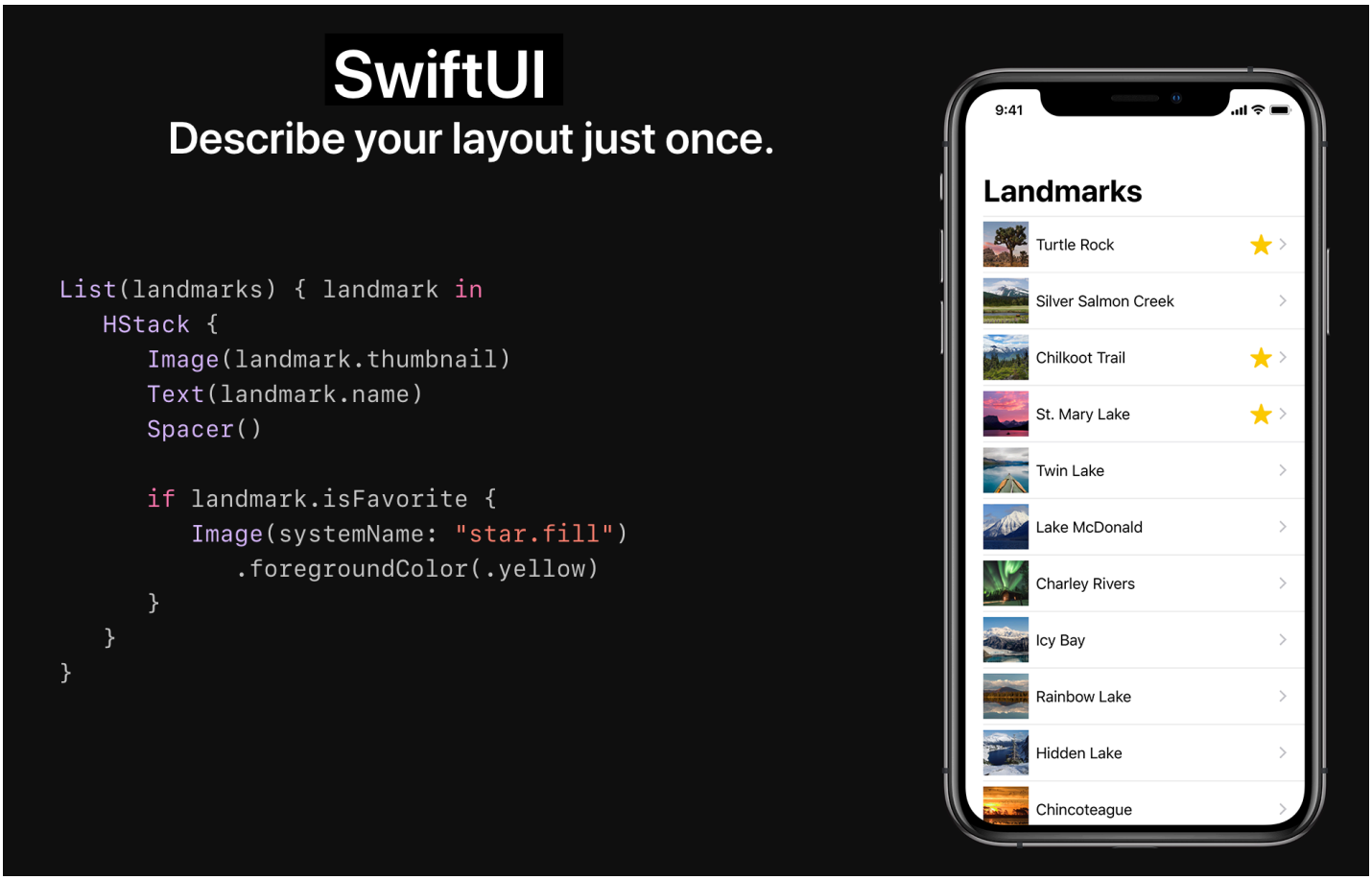
De esta forma, la ejecución de un programa declarativo tiene que ver más con algún modelo formal o matemático que con un programa tradicional imperativo. Define un conjunto de reglas y definiciones *de estilo matemático*.

La programación declarativa no es exclusiva de los lenguajes funcionales. Existen muchos lenguajes no funcionales con características declarativas. Por ejemplo Prolog, en el que un programa se define como un

conjunto de reglas lógicas y su ejecución realiza una deducción lógica matemática que devuelve un resultado. En dicha ejecución no son relevantes los pasos internos que realiza el sistema sino las relaciones lógicas entre los datos y los resultados finales.

Un ejemplo claro de programación declarativa es una **hoja de cálculo**. Las celdas contiene valores o expresiones matemáticas que se actualizan automáticamente cuando cambiamos los valores de entrada. La relación entre valores y resultados es totalmente matemática y para su cálculo no tenemos que tener en cuenta pasos de ejecución. Evidente, por debajo de la hoja de cálculo existe un programa que realiza el su cálculo de la hoja, pero cuando estamos usándola no nos preocupa esa implementación. Podemos no preocuparnos de ella y usar únicamente el modelo matemático definido en la hoja.

Otro ejemplo muy actual de programación declarativa es SwiftUI, el nuevo API creado por Apple para definir las interfaces de usuario de las aplicaciones iOS.



En el código de la imagen vemos una descripción de cómo está definida la aplicación: una lista de lugares (*landmarks*) apilada verticalmente. Para cada lugar se define su imagen, su texto, y una estrella si el lugar es favorito.

El código es declarativo porque no hay pasos de ejecución para definir la interfaz. No existe un bucle que va añadiendo elementos a la interfaz. Vemos una declaración de cómo la interfaz va estar definida. El compilador del lenguaje y el API son los responsables de construir esa declaración y mostrar la interfaz tal y como nosotros queremos.

1.3.1.1. DECLARACIÓN DE FUNCIONES

La programación funcional utiliza un estilo de programación declarativo. Declaramos funciones en las que

se transforman unos datos de entrada en unos datos de salida. Veremos que esta transformación se realiza mediante la evaluación de expresiones, sin definir valores intermedios, ni variables auxiliares, ni pasos de ejecución. Únicamente se van componiendo llamadas a funciones auxiliares que construyen el valor resultante.

Tal y como ya hemos visto, el siguiente ejemplo es una **declaración** en Scheme de una función que toma como entrada un número y devuelve su cuadrado:

```
(define (cuadrado x)
   (* x x))
```

En el cuerpo de la función `cuadrado` vemos que no se utiliza ninguna variable auxiliar, sino que únicamente se llama a la función `*` (multiplicación) pasando el valor de `x`. El valor resultante es el que se devuelve.

Por ejemplo, si llamamos a la función pasándole el parámetro `4` devuelve el resultado de multiplicar 4 por si mismo, 16.

```
(cuadrado 4) ; ⇒ 16
```

1.3.2. Programación imperativa

Repasemos un algunas características propias de la programación imperativa **que no existen en la programación funcional**. Son características a las que estamos muy habituados porque son propias de los lenguajes más populares y con los que hemos aprendido a programar (C, C++, Java, python, etc.)

- Pasos de ejecución
- Mutación
- Efectos laterales
- Estado local mutable en las funciones

Veremos que, aunque parece imposible, es posible programar sin utilizar estas características. Lo demuestran lenguajes de programación funcional como Haskell, Clojure o el propio Scheme.

1.3.2.1. PASOS DE EJECUCIÓN

Una de las características básicas de la programación imperativa es la utilización de pasos de ejecución. Por ejemplo, en C podemos realizar los siguientes pasos de ejecución:

```
int a = cuadrado(8);
int b = doble(a);
int c = cuadrado(b);
return c
```

O, por ejemplo, si queremos filtrar y procesar una lista de pedidos en Swift podemos hacerlo en dos sentencias:

```
filtrados = filtra(pedidos);
procesados = procesa(filtrados);
```

```
return procesados;
```

Sin embargo, en programación funcional (por ejemplo, Scheme) no existen pasos de ejecución separados por sentencias. Como hemos visto antes, la forma típica de expresar las instrucciones anteriores es componer todas las operaciones en una única instrucción:

```
(cuadrado (doble (cuadrado 8))) ; ⇒ 16384
```

El segundo ejemplo lo podemos componer de la misma forma:

```
(procesa (filtra pedidos))
```

1.3.2.2. MUTACIÓN

En los lenguajes imperativos es común modificar el valor de las variables en los pasos de ejecución:

```
int x = 10;
int x = x + 1;
```

La expresión `x = x + 1` es una expresión de asignación que modifica el valor anterior de una variable por un nuevo valor. El *estado* de las variables (su valor) cambia con la ejecución de los pasos del programa.

A esta asignación que modifica un valor ya existente se le denomina *asignación destructiva* o **mutación**.

En programación imperativa también se puede modificar (mutar) el valor de componentes de estructuras de datos, como posiciones de un array, de una lista o de un diccionario.

En programación funcional, por contra, **las definiciones son inmutables**, y una vez asignado un valor a un identificador no se puede modificar éste. En programación funcional **no existe sentencia de asignación** que pueda modificar un valor ya definido. Se entienden las variables como variables matemáticas, no como referencias a una posiciones de memoria que puede ser modificada.

Por ejemplo, la forma especial `define` en Scheme crea un nuevo identificador y le da el valor definido de forma permanente. Si escribimos el siguiente código en un programa en Scheme:

```
#lang racket

(define a 12)
(define a 200)
```

tendremos el siguiente error:

```
module: identifier already defined in: a
```

> **Nota**
>
> En el intérprete REPL del DrRacket sí que podemos definir más de una vez la misma función o identificador. Se ha diseñado así para facilitar el uso del intérprete para la prueba de expresiones en Scheme.

En los lenguajes de programación imperativos es habitual introducir también sentencias declarativas. Por ejemplo, en el siguiente código Java las líneas 1 y 3 las podríamos considerar declarativas y las 2 y 4 imperativas:

```
1. int x = 1;
2. x = x+1;
3. int y = x+1;
4. y = x;
```

1.3.2.3. MUTACIÓN Y EFECTOS LATERALES

En programación imperativa es habitual también trabajar con referencias y hacer que más de un identificador referencie el mismo valor. Esto produce la posibilidad de que la mutación del valor a través de uno de los identificadores produzca un **efecto lateral** (*side effect* en inglés) en el que el valor de un identificador cambia sin ejecutar ninguna expresión en la que se utilice explícitamente el propio identificador.

Por ejemplo, en la mayoría de lenguajes orientados a objetos los identificadores guardan referencias a objetos. De forma que si asignamos un objeto a más de un identificador, todos los identificadores están accediendo al mismo objeto. Si mutamos algún valor del objeto a través de un identificador provocamos un efecto lateral en los otros identificadores.

Por ejemplo, lo siguiente es un ejemplo de una mutación en programación imperativa, en la que se modifican los atributos de un objeto en Java:

```
Point2D p1 = new Point2D(3.0, 2.0); // creamos un punto 2D con coordX=3.0 y coordY=2.0
p1.getCoordX(); // la coord x de p2 es 3.0
p1.setCoordX(10.0);
p1.getCoordX(); // la coord x de p1 es 10.0
```

Si el objeto está asignado a más de una variable tendremos el **efecto lateral** (*side effect*) en el que el dato guardado en una variable cambia después de una sentencia en la que no se ha usado esa variable:

```
Point2D p1 = new Point2D(3.0, 2.0); // la coord x de p1 es 3.0
p1.getCoordX(); // la coord x de p1 es 3.0
Point2D p2 = p1;
p2.setCoordX(10.0);
p1.getCoordX(); // la coord x de p1 es 10.0, sin que ninguna sentencia haya modificado
directamente p1
```

El mismo ejemplo anterior, en C:

```
typedef struct {
  float x;
  float y;
}TPunto;

TPunto p1 = {3.0, 2.0};
printf("Coordenada x: %f", p1.x);  // 3.0
TPunto *p2 = &p1;
p2->x = 10.0;
printf("Coordenada x: %f", p1.x);  // 10.0 Efecto lateral
```

Los efectos laterales son los responsables de muchos *bugs* y hay que ser muy consciente de su uso. Son especialmente complicados de depurar los *bugs* debidos a efectos laterales en programas concurrentes con múltiples hilos de ejecución, en los que varios hilos pueden acceder a las mismas referencias y provocar condiciones de carrera.

Por otro lado, también existen situaciones en las que su utilización permite ganar mucha eficiencia porque podemos definir estructuras de datos en el que los valores son compartidos por varias referencias y modificando un único valor se actualizan de forma instantánea esas referencias.

En los lenguajes en los que no existe la mutación no se producen efectos laterales, ya que no es posible modificar el valor de una variable una vez establecido. Los programas que escribamos en estos lenguajes van a estar libres de este tipo de *bugs* y van a poder ser ejecutado sin problemas en hilos de ejecución concurrente.

Por otro lado, la ausencia de mutación hace que sean algo más costosas ciertas operaciones, como la construcción de estructuras de datos nuevas a partir de estructuras ya existentes. Veremos, por ejemplo, que la única forma de añadir un elemento al final de una lista será construir una lista nueva con todos los elementos de la lista original y el nuevo elemento. Esta operación tiene un coste lineal con el número de elementos de la lista. Sin embargo, en una lista en la que pudiéramos utilizar la mutación podríamos implementar esta operación con coste constante.

1.3.2.4. ESTADO LOCAL MUTABLE

Otra característica de la programación imperativa es lo que se denomina **estado local mutable** en funciones, procedimientos o métodos. Se trata la posibilidad de que una invocación a un método o una función modifique un cierto estado, de forma que la siguiente invocación devuelva un valor distinto. Es una característica básica de la programación orientada a objetos, donde los objetos guardan valores que se modifican con la invocaciones a sus métodos.

Por ejemplo, en Java, podemos definir un contador que incrementa su valor:

```
public class Contador {
    int c;

    public Contador(int valorInicial) {
        c = valorInicial;
    }

    public int valor() {
        c++;
        return c;
    }
}
```

Cada llamada al método `valor()` devolverá un valor distinto:

```
Contador cont = new Contador(10);
cont.valor(); // 11
cont.valor(); // 12
cont.valor(); // 13
```

También se pueden definir funciones con estado local mutable en C:

```
int function contador () {
    static int c = 0;

    c++;
    return c;
}
```

Cada llamada a la función `contador()` devolverá un valor distinto:

```
contador() ;; 1
contador() ;; 2
contador() ;; 3
```

Por el contrario, los lenguajes funcionales tienen la propiedad de **transparencia referencial**: es posible sustituir cualquier aparición de una expresión por su resultado sin que cambia el resultado final del programa. Dicho de otra forma, en programación funcional, **una función siempre devuelve el mismo valor cuando se le llama con los mismos parámetros**. Las funciones no modifican ningún estado, no acceden a ninguna variable ni objeto global y modifican su valor.

1.3.2.5. RESUMEN

Un resumen de las características fundamentales de la programación declarativa frente a la programación imperativa. En los siguientes apartados explicaremos más estas características.

Características de la programación declarativa

- Variable = nombre dado a un valor (declaración)
- En lugar de pasos de ejecución se utiliza la composición de funciones
- No existe asignación ni cambio de estado
- No existe mutación, se cumple la *transferencia referencial*: dentro de un mismo ámbito todas las ocurrencias de una variable y las llamadas a funciones devuelven el mismo valor

Características de la programación imperativa

- Variable = nombre de una zona de memoria
- Asignación
- Referencias
- Pasos de ejecución

1.4. Modelo de computación de sustitución

Un modelo computacional es un formalismo (conjunto de reglas) que definen el funcionamiento de un programa. En el caso de los lenguajes funcionales basados en la evaluación de expresiones, el modelo computacional define cuál será el resultado de evaluar una expresión.

El **modelo de sustitución** es un modelo muy sencillo que permite definir la semántica de la evaluación de expresiones en lenguajes funcionales como Scheme. Se basa en una versión simplificada de la regla de reducción del cálculo lambda.

Es un modelo basado en la reescritura de unos términos por otros. Aunque se trata de un modelo abstracto, sería posible escribir un intérprete que, basándose en este modelo, evalúe expresiones funcionales.

Supongamos un conjunto de definiciones en Scheme:

```
(define (doble x)
    (+ x x))

(define (cuadrado y)
    (* y y))

(define (f z)
    (+ (cuadrado (doble z)) 1))

(define a 2)
```

Supongamos que, una vez realizadas esas definiciones, se evalúa la siguiente expresión:

```
(f (+ a 1))
```

¿Cuál será su resultado? Si lo hacemos de forma intuitiva podemos pensar que `37`. Si lo comprobamos en el intérprete de Scheme veremos que devuelve 37. ¿Hemos seguido algunas reglas específicas? ¿Qué reglas son las que sigue el intérprete? ¿Podríamos implementar nosotros un intérprete similar? Sí, usando las reglas del modelo de sustitución.

El modelo de sustitución define cuatro reglas sencillas para evaluar una expresión. Llamemos a la expresión *e*. Las reglas son las siguientes:

1. Si *e* es un valor primitivo (por ejemplo, un número), devolvemos ese mismo valor.
2. Si *e* es un identificador, devolvemos su valor asociado con un `define` (se lanzará un error si no existe ese valor).
3. Si *e* es una expresión del tipo *(f arg1 ... argn)*, donde *f* es el nombre de una función primitiva (`+`, `-`, ...), evaluamos uno a uno los argumentos *arg1 ... argn* (con estas mismas reglas) y evaluamos la función primitiva con los resultados.

La regla 4 tiene dos variantes, dependiendo del orden de evaluación que utilizamos.

Orden aplicativo

4. Si *e* es una expresión del tipo *(f arg1 ... argn)*, donde *f* es el nombre de una función definida con un `define`, tenemos que evaluar primero los argumentos *arg1 ... argn* y después **sustituir *f* por su cuerpo**, reemplazando cada parámetro formal de la función por el correspondiente **argumento evaluado**. Después evaluaremos la expresión resultante usando estas mismas reglas.

Orden normal

4. Si *e* es una expresión del tipo *(f arg1 ... argn)*, donde *f* es el nombre de una función definida con un `define`, tenemos que **sustituir *f* por su cuerpo**, reemplazando cada parámetro formal de la función por el correspondiente **argumento sin evaluar**. Después evaluar la expresión resultante usando estas mismas reglas.

En el orden aplicativo se realizan las evaluaciones antes de realizar las sustituciones, lo que define una evaluación de *dentro a fuera* de los paréntesis. Cuando se llega a una expresión primitiva se evalúa.

En el orden normal se realizan todas las sustituciones hasta que se tiene una larga expresión formada por expresiones primitivas; se evalúa entonces.

Ambas formas de evaluación darán el mismo resultado en programación funcional. Scheme utiliza el orden aplicativo.

1.4.1. Ejemplo 1

Vamos a empezar con un ejemplo sencillo para comprobar cómo se evalúa una misma expresión utilizando ambos modelos de sustitución. Supongamos las siguientes definiciones:

```
(define (doble x)
    (+ x x))

(define (cuadrado y)
    (* y y))

(define a 2)
```

Queremos evaluar la siguiente expresión:

```
(doble (cuadrado a))
```

La evaluación utilizando el **modelo de sustitución aplicativo**, usando paso a paso las reglas anteriores, es la siguiente (en cada línea se indica entre paréntesis la regla usada):

```
(doble (cuadrado a)) ⇒       ; Sustituimos a por su valor (R2)
(doble (cuadrado 2)) ⇒       ; Sustitumos cuadrado por su cuerpo (R4)
(doble (* 2 2)) ⇒            ; Evaluamos (* 2 2) (R3)
(doble 4) ⇒                  ; Sustituimos doble por su cuerpo (R4)
(+ 4 4) ⇒                    ; Evaluamos (+ 4 4) (R3)
8
```

Podemos comprobar que en el modelo aplicativo se intercalan las sustituciones de una función por su cuerpo (regla 4) y las evaluaciones de expresiones (regla 3).

Por el contrario, la evaluación usando el **modelo de sustitución normal** es:

```
(doble (cuadrado a)) ⇒            ; Sustituimos doble por su cuerpo (R4)
(+ (cuadrado a) (cuadrado a) ⇒    ; Sustituimos cuadrado por su cuerpo (R4)
(+ (* a a) (* a a)  ⇒             ; Sustitumos a por su valor (R2)
```

```
(+ (* 2 2) (* 2 2))  ⇒            ; Evaluamos (* 2 2) (R3)
(+ 4 (* 2 2))  ⇒                  ; Evaluamos (* 2 2) (R3)
(+ 4 4)  ⇒                        ; Evaluamos (+ 4 4) (R3)
8
```

Al usar este modelo de evaluación primero se realizan todas las sustituciones (regla 4) y después todas las evaluaciones (regla 3).

Las sustituciones se hacen de izquierda a derecha (de fuera a dentro de los paréntesis). Primero se sustituye `doble` por su cuerpo y después `cuadrado`.

1.4.2. Ejemplo 2

Veamos la evaluación del ejemplo algo más complicado que hemos planteado al comienzo:

```
(define (doble x)
    (+ x x))

(define (cuadrado y)
    (* y y))

(define (f z)
    (+ (cuadrado (doble z)) 1))

(define a 2)
```

Expresión a evaluar:

```
(f (+ a 1))
```

Resultado de la evaluación usando el **modelo de sustitución aplicativo**:

```
(f (+ a 1)) ⇒                ; Para evaluar f, evaluamos primero su argumento (+ a 1)
(R4)
                             ; y sustituimos a por 2 (R2)
(f (+ 2 1)) ⇒                ; Evaluamos (+ 2 1) (R3)
(f 3) ⇒                      ; (R4)
(+ (cuadrado (doble 3)) 1) ⇒ ; Sustituimos (doble 3) (R4)
(+ (cuadrado (+ 3 3)) 1) ⇒   ; Evaluamos (+ 3 3) (R3)
(+ (cuadrado 6) 1) ⇒         ; Sustitumos (cuadrado 6) (R4)
(+ (* 6 6) 1) ⇒              ; Evaluamos (* 6 6) (R3)
(+ 36 1) ⇒                   ; Evaluamos (+ 36 1) (R3)
37
```

Y veamos el resultado de usar el **modelo de sustitución normal**:

```
(f (+ a 1)) ⇒                       ; Sustituimos (f (+ a 1))
                                    ; por su definición, con z = (+ a 1) (R4)
(+ (cuadrado (doble (+ a 1))) 1) ⇒  ; Sustituimos (cuadrado ...) (R4)
(+ (* (doble (+ a 1))
      (doble (+ a 1))) 1)           ; Sustituimos (doble  ...) (R4)
(+ (* (+ (+ a 1) (+ a 1))
      (+ (+ a 1) (+ a 1))) 1) ⇒     ; Evaluamos a (R2)
(+ (* (+ (+ 2 1) (+ 2 1))
```

```
      (+ (+ 2 1) (+ 2 1))) 1) ⇒    ; Evaluamos (+ 2 1) (R3)
(+ (* (+ 3 3)
      (+ 3 3)) 1) ⇒                ; Evaluamos (+ 3 3) (R3)
(+ (* 6 6) 1) ⇒                    ; Evaluamos (* 6 6) (R3)
(+ 36 1) ⇒                         ; Evaluamos (+ 36 1) (R3)
37
```

En programación funcional el resultado de evaluar una expresión es el mismo independientemente del tipo de orden. Pero si estamos fuera del paradigma funcional y las funciones tienen estado y cambian de valor entre distintas invocaciones sí que importan si escogemos un orden.

Por ejemplo, supongamos una función `(random x)` que devuelve un entero aleatorio entre 0 y *x*. Esta función no cumpliría el paradigma funcional, porque devuelve un valor distinto con el mismo parámetro de entrada.

Evaluamos las siguientes expresiones con orden aplicativo y normal, para comprobar que el resultado es distinto.

```
(define (zero x) (- x x))
(zero (random 10))
```

Si evaluamos la última expresión en orden aplicativo:

```
(zero (random 10)) ⇒ ; Evaluamos (random 10) (R3)
(zero 3) ⇒           ; Sustituimos (zero ...) (R4)
(- 3 3) ⇒            ; Evaluamos - (R3)
0
```

Si lo evaluamos en orden normal:

```
(zero (random 10)) ⇒            ; Sustituimos (zero ...) (R4)
(- (random 10) (random 10)) ⇒   ; Evaluamos (random 10) (R3)
(- 5 3) ⇒                       ; Evaluamos - (R3)
2
```

2. Scheme como lenguaje de programación funcional

Ya hemos visto cómo definir funciones y evaluar expresiones en Scheme. Vamos continuar con ejemplos concretos de otras características funcionales características funcionales de Scheme.

En concreto, veremos:

- Símbolos y primitiva `quote`
- Uso de listas
- Definición de funciones recursivas en Scheme

2.1. Funciones y formas especiales

En el seminario de Scheme hemos visto un conjunto de primitivas que podemos utilizar en Scheme.

Podemos clasificar las primitivas en **funciones** y **formas especiales**. Las funciones se evalúan usando el modelo de sustitución aplicativo ya visto:

- Primero se evalúan los argumentos y después se sustituye la llamada a la función por su cuerpo y se vuelve a evaluar la expresión resultante.
- Las expresiones siempre se evalúan desde los paréntesis interiores a los exteriores.

Las *formas especiales* son expresiones primitivas de Scheme que tienen una forma de evaluarse propia, distinta de las funciones.

2.2. Formas especiales en Scheme

Veamos la forma de evaluar las distintas formas especiales en Scheme. En estas formas especiales no se aplica el modelo de sustitución, al no ser invocaciones de funciones, sino que cada una se evalúa de una forma diferente.

2.2.1. Forma especial `define`

Sintaxis

```
(define <identificador> <expresión>)
```

Evaluación

1. Evaluar *expresión*
2. Asociar el valor resultante con el *identificador*

Ejemplo

```
(define base 10)   ; Asociamos a 'base' el valor 10
(define altura 12) ; Asociamos a 'altura' el valor 12
(define area (/ (* base altura) 2)) ; Asociamos a 'area' el valor 60
```

2.2.2. Forma especial `define` para definir funciones

Sintaxis

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

Evaluación

La semana que viene veremos con más detalle la semántica, y explicaremos la forma especial `lambda` que es la que realmente crea la función. Hoy nos quedamos en la siguiente descripción de alto nivel de la semántica:

1. Crear la función con el *cuerpo*

2. Dar a la función el nombre *nombre-función*

Ejemplo

```
(define (factorial x)
    (if (= x 0)
        1
        (* x (factorial (- x 1)))))
```

2.2.3. Forma especial `if`

Sintaxis

```
(if <condición> <expresión-true> <expresión-false>)
```

Evaluación

1. Evaluar *condición*
2. Si el resultado es `#t` evaluar la *expresión-true*, en otro caso, evaluar la *expresión-false*

Ejemplo

```
(if (> 10 5) (substring "Hola qué tal" (+ 1 1) 4) (/ 12 0))

;; Evaluamos (> 10 5). Como el resultado es #t, evaluamos
;; (substring "Hola qué tal" (+ 1 1) 4), que devuelve "la"
```

Nota

Al ser `if` una forma especial, no se evalúa utilizando el modelo de sustitución, sino usando las reglas propias de la forma especial.

Por ejemplo, veamos la siguiente expresión:

```
(if (> 3 0) (+ 2 3) (/ 1 0)) ; ⇒ 5
```

Si se evaluara con el modelo de sustitución se lanzaría un error de división por cero al intentar evaluar `(/ 1 0)`. Sin embargo, esa expresión no llega a evaluarse, porque la condición `(> 3 0)` es cierta y sólo se evalúa la suma `(+ 2 3)`.

2.2.4. Forma especial `cond`

Sintaxis

```
(cond
    (<exp-cond-1> <exp-consec-1>)
    (<exp-cond-2> <exp-consec-2>)
    ...
    (else <exp-consec-else>))
```

Evaluación

1. Se evalúan de forma ordenada todas las *exp-cond-i* hasta que una de ellas devuelva `#t`
2. Si alguna *exp-cond-i* devuelve `#t`, se devuelve el valor de la *exp-consec-i*.
3. Si ninguna *exp-cond-i* es cierta, se devuelve el valor resultante de evaluar *exp-consec-else*.

Ejemplo

```
(cond
   ((> 3 4) "3 es mayor que 4")
   ((< 2 1) "2 es menor que 1")
   ((= 3 1) "3 es igual que 1")
   ((> 3 5) "3 es mayor que 2")
   (else "ninguna condición es cierta"))

;; Se evalúan una a una las expresiones (> 3 4),
;; (< 2 1), (= 3 1) y (> 3 5). Como ninguna de ella
;; es cierta se devuelve la cadena "ninguna condición es cierta".
```

2.2.4.1. Formas especiales `and` y `or`

Las expresiones lógicas `and` y `or` no son funciones, sino formas especiales. Lo podemos comprobar con el siguiente ejemplo:

```
(and #f (/ 3 0)) ; ⇒ #f
(or #t (/ 3 0))  ; ⇒ #t
```

Si `and` y `or` fueran funciones, seguirían la regla que hemos visto de evaluar primero los argumentos y después invocar a la función con los resultados. Esto produciría un error al evaluar la expresión `(/ 3 0)`, al ser una división por 0.

Sin embargo, vemos que las expresiones no dan error y devuelven un valor booleano. ¿Por qué? Porque `and` y `or` no son funciones, sino formas especiales que se evalúan de forma diferente a las funciones.

En concreto, `and` y `or` van evaluando los argumentos hasta que encuentran un valor que hace que ya no sea necesario evaluar el resto.

Sintaxis

```
(and exp1 ... expn)
(or exp1 ... expn)
```

Evaluación `and`

- Se evalúa la expresión 1. Si el resultado es `#f`, se devuelve `#f`, en otro caso, se evalúa la siguiente expresión.
- Se repite hasta la última expresión, cuyo resultado se devuelve.

Ejemplos `and`

```
(and #f (/ 3 0)) ; ⇒ #f
(and #t (> 2 1) (< 5 10)) ; ⇒ #t
(and #t (> 2 1) (< 5 10) (+ 2 3)) ; ⇒ 5
```

La regla de evaluación de `and` hace que sea posible que devuelva resultados no booleanos, como el último ejemplo. Sin embargo, no es recomendable usarlo de esta forma y en la asignatura no lo vamos a hacer nunca.

Evaluación `or`

- Se evalúa la expresión 1. Si el resultado es distinto de `#f` se devuelve ese resultado. Si el resultado es `#f` se evalúa la siguiente expresión.
- Se repite hasta la última expresión, cuyo resultado se devuelve.

Ejemplos `or`

```
(or #t (/ 3 0)) ; ⇒ #t
(or #f (< 2 10) (> 5 10)) ; ⇒ #t
(or (+ 2 3) (> 5 10)) ; ⇒ 5
```

Al igual que `and`, la regla de evaluación de `or` hace que sea posible que devuelva resultados no booleanos, como el último ejemplo. Tampoco es recomendable usarlo de esta forma.

2.3. Forma especial `quote` y símbolos

Sintaxis

```
(quote <identificador>)
```

Evaluación

- Se devuelve el identificador sin evaluar (un símbolo).
- Se abrevia en con el carácter `'`.

Ejemplos

```
(quote x) ; el símbolo x
'hola ; el símbolo hola
```

A diferencia de los lenguajes imperativos, Scheme trata a los *identificadores* (nombres que se les da a las variables) como datos del lenguaje de tipo **symbol**. En el paradigma funcional a los identificadores se les denomina *símbolos*.

Los símbolos son distintos de las cadenas. Una cadena es un tipo de dato **compuesto** y se guardan en memoria todos y cada uno de los caracteres que la forman. Sin embargo, los símbolos son tipos atómicos, que se representan en memoria con un único valor determinado por el *código hash* del identificador.

Ejemplos de funciones Scheme con símbolos:

```
(define x 12)
(symbol? 'x) ; ⇒ #t
(symbol? x) ; ⇒ #f ¿Por qué?
(symbol? 'hola-que<>)
(symbol->string 'hola-que<>)
'mañana
'lápiz ; aunque sea posible, no vamos a usar acentos en los símbolos
; pero sí en los comentarios
(symbol? "hola") ; #f
(symbol?  #f) ; #f
(symbol? (first '(hola cómo estás))) ; #t
(equal? 'hola 'hola)
(equal? 'hola "hola")
```

Como hemos visto anteriormente, un símbolo puede asociarse o ligarse (*bind*) a un valor (cualquier dato *de primera clase*) con la forma especial `define`.

```
(define pi 3.14159)
```

> **Nota**
>
> No es correcto escribir `(define 'pi 3.14156)` porque la forma especial `define` debe recibir un identificador *sin quote*.

Cuando escribimos un símbolo en el prompt de Scheme el intérprete lo evalúa y devuelve su valor:

```
> pi
3.14159
```

Los nombres de las funciones (`equal?`, `sin`, `+`, ...) son también símbolos y Scheme también los evalúa (en un par de semanas hablaremos de las funciones como objetos primitivos en Scheme):

```
> sin
#<procedure:sin>
> +
#<procedure:+>
> (define (cuadrado x) (* x x))
> cuadrado
#<procedure:cuadrado>
```

Los símbolos son tipos primitivos del lenguaje: pueden pasarse como parámetros o ligarse a variables.

```
> (define x 'hola)
> x
hola
```

2.4. Forma expecial `quote` con expresiones

Sintaxis

```
(quote <expresión>)
```

Evaluación

Si `quote` recibe una expresión correcta de Scheme (una expresión entre paréntesis) se devuelve la lista o pareja pareja definida por la expresión (sin evaluar sus elementos).

Ejemplos

```
'(1 2 3) ; ⇒ (1 2 3) Una lista
'(+ 1 2 3 4) ; La lista formada por el símbolo + y los números 1 2 3 4
(quote (1 2 3 4)) ; La lista formada por los números 1 2 3 4
'(a b c) ; ⇒ La lista con los símbolos a, b, y c
'(* (+ 1 (+ 2 3)) 5) ; Una lista con 3 elementos, el segundo de ellos otra lista
'(1 . 2) ; ⇒ La pareja (1 . 2)
'((1 . 2) (2 . 3)) ; ⇒ Una lista con las parejas (1 . 2) y (2 . 3)
```

2.5. Función `eval`

Una vez vista la forma especial `quote` podemos explicar la función `eval`. La función `eval` es una instrucción muy curiosa de los lenguajes funcionales. Permite invocar al intérprete en tiempo de ejecución y hacer que éste evalúe una expresión que puede haberse construido dinámicamente.

Sintaxis

```
(eval <expresión>)
```

Evaluación

La función `eval` invoca al intérprete para realizar la evaluación de la expresión que se le pasa como parámetro y devuelve el resultado de dicha evaluación.

Ejemplos

```
(define a 10)
(eval 'a) ; ⇒ 10

(eval '(+ 1 2 3)) ; ⇒ 6

(define lista (list '+ 1 2 3))
(eval lista) ; ⇒ 6

(define a 10)
(define x 'a)
(eval 'x) ; ⇒ a
(eval x) ; ⇒ 10
(eval (eval 'x)) ; ⇒ 10
```

Una nota sobre la evaluación de eval

Al ser `eval` una función, la expresión que se le pasa como parámetro se evalúa previamente, antes de ser

procesada por `eval`. En el caso de ser una expresión con `quote`, el resultado de la evaluación es la propia expresión (una lista), que es procesada por `eval`.

Por ejemplo, en la siguiente expresión

```
(eval (+ 2 3)) ; ⇒ 5
```

primero se evaluaría la expresión `(+ 2 3)` y lo que se le pasaría a `eval` sería un `5`. El resultado de evaluar un `5` sería un `5`.

Sin embargo, en la siguiente expresión:

```
(eval '(+ 2 3))
```

el resultado de evaluar `'(+ 2 3)` devolvería la lista `(+ 2 3)` que es la que se pasaría a `eval`. El resultado de evaluar esa expresión también sería `5`.

2.6. Listas

Otra de las características fundamentales del paradigma funcional es la utilización de listas. Ya hemos visto en el seminario de Scheme las funciones más importantes para trabajar con ellas. Vamos a repasarlas de nuevo en este apartado, antes de ver algún ejemplo de cómo usar la recursión con listas.

Ya hemos visto en dicho seminario que Scheme es un lenguaje débilmente tipado. Una variable o parámetro no se declara de un tipo y puede contener cualquier valor. Sucede igual con las listas: una lista en Scheme puede contener cualquier valor, incluyendo otras listas.

2.6.1. Diferencia entre la función `list` y la forma especial `quote`

En el seminario de Scheme explicamos que podemos crear listas de forma dinámica, llamando a la función `list` y pasándole un número variable de parámetros que son los elementos que se incluirán en la lista:

```
(list 1 2 3 4 5) ; ⇒ (1 2 3 4)
(list 'a 'b 'c) ; ⇒ (a b c)
(list 1 'a 2 'b 3 'c #t) ; ⇒ (1 a 2 b 3 c #t)
(list 1 (+ 1 1) (* 2 (+ 1 2))) ; ⇒ (1 2 6)
```

Las expresiones interiores se evalúan y se llama a la función `list` con los valores resultantes.

Otro ejemplo:

```
(define a 1)
(define b 2)
(define c 3)
(list a b c) ; ⇒ (1 2 3)
```

Como hemos visto cuando hemos hablado de `quote`, esta forma especial también puede construir una lista. Pero lo hace sin evaluar sus elementos.

Por ejemplo:

```
'(1 2 3 4) ; ⇒ (1 2 3 4)
(define a 1)
(define b 2)
(define c 3)
'(a b c) ; ⇒ (a b c)
'(1 (+ 1 1) (* 2 (+ 1 2))) ; ⇒ (1 (+ 1 1) (* 2 (+ 1 2)))
```

La última lista tiene 3 elementos:

- El número 1
- La lista `(+ 1 1)`
- La lista `(* 2 (+ 1 2))`

Es posible definir una lista vacía (sin elementos) realizando una llamada sin argumentos a la función `list` o utilizando el símbolo `():

```
(list) ; ⇒ ()
`() ; ⇒ ()
```

La diferencia entre creación de listas con la función `list` y con la forma especial `quote` se puede comprobar en los ejemplos.

La evaluación de la función `list` funciona como cualquier función, primero se evalúan los argumentos y después se invoca a la función con los argumentos evaluados. Por ejemplo, en la siguiente invocación se obtiene una lista con cuatro elementos resultantes de las invocaciones de las funciones dentro del paréntesis:

```
(list 1 (/ 2 3) (+ 2 3)) ; ⇒ (1 2/3 5)
```

Sin embargo, usamos `quote` obtenemos una lista con sublistas con símbolos en sus primeras posiciones:

```
'(1 (/ 2 3) (+ 2 3)) ; ⇒ (1 (/ 2 3) (+ 2 3))
```

2.6.2. Selección de elementos de una lista: `first` y `rest`

En el seminario vimos también cómo obtener los elementos de una lista.

- Primer elemento: función `first`
- Resto de elementos: función `rest` (los devuelve en forma de lista)

Ejemplos:

```
(define lista1 '(1 2 3 4))
(first lista1) ; ⇒ 1
(rest lista1) ; ⇒ (2 3 4)
(define lista2 '((1 2) 3 4))
(first lista2) ⇒ (1 2)
```

```
(rest lista2) ⇒ (3 4)
```

2.6.3. Composición de listas: `cons` y `append`

Por último, en el seminario vimos también cómo crear nuevas listas a partir de ya existentes con las funciones `cons` y `append`.

La función `cons` crea una lista nueva resultante de añadir un elemento al comienzo de la lista. Esta función es la forma habitual de construir nuevas listas a partir de una lista ya existente y un nuevo elemento.

```
(cons 1 '(1 2 3 4)) ; ⇒ (1 1 2 3 4)
(cons 'hola '(como estás)) ; ⇒ (hola como estás)
(cons '(1 2) '(1 2 3 4))  ; ⇒ ((1 2) 1 2 3 4)
```

La función `append` se usa para crear una lista nueva resultado de concatenar dos o más listas

```
(define list1 '(1 2 3 4))
(define list2 '(hola como estás))
(append list1 list2) ; ⇒ (1 2 3 4 hola como estás)
```

2.7. Recursión

Otra característica fundamental de la programación funcional es la no existencia de bucles. Un bucle implica la utilización de pasos de ejecución en el programa y esto es característico de la programación imperativa.

En programación funcional las iteraciones se realizan con recursión.

2.7.1. Función `(suma-hasta x)`

Por ejemplo, podemos definir la función `(suma-hasta x)` que devuelve la suma de los números hasta el parámetro `x` cuyo valor pasamos en la invocación de la función.

Por ejemplo, `(suma-hasta 5)` devolverá `0+1+2+3+4+5 = 15`.

La definición de la función es la siguiente:

```
(define (suma-hasta x)
   (if (= 0 x)
      0
      (+ (suma-hasta (- x 1)) x)))
```

En una definición recursiva siempre tenemos un **caso general** y un **caso base**. El caso base define el valor que devuelve la función en el caso elemental en el que no hay que hacer ningún cálculo. El caso general define una expresión que contiene una llamada a la propia función que estamos definiendo.

El **caso base** es el caso en el que `x` vale 0. En este caso devolvemos el propio 0, no hay que realizar ningún cálculo.

El **caso general** es en el que se realiza la llamada recursiva. Esta llamada devuelve un valor que se utiliza

para cálculo final evaluando la expresión del caso general con valores concretos.

En programación funcional, al no existir efectos laterales, lo único que importa cuando realizamos una recursión es el valor devuelto por la llamada recursiva. Ese valor devuelto se combina con el resto de la expresión del caso general para construir el valor resultante.

> **Importante**
>
> Para entender la recursión no es conveniente utilizar el depurador, ni hacer trazas, ni *entrar en la recursión*, sino que hay que suponer que **la llamada recursiva se ejecuta y devuelve el valor que debería. ¡Debemos confiar en la recursión!**.

El caso general del ejemplo anterior indica lo siguiente:

```
Para calcular la suma hasta x:
    Llamamos a la recursión para que calcule la suma hasta x-1
    (confiamos en que la implementación funciona bien y esta llamada
    nos devolverá el resultado hasta x-1) y a ese resultado le sumamos
    el propio número x.
```

Siempre es aconsejable usar un ejemplo concreto para probar el caso general. Por ejemplo, el caso general de la suma hasta 5 se calculará de la siguiente forma:

```
(+ (suma-hasta (- 5 1)) 5) ; ⇒
(+ (suma-hasta 4) 5) ;  ⇒ confiamos en la recursión:
                     ;    (suma-hasta 4) = 4+3+2+1 = 10  ⇒
(+ 10 5) ; ⇒
15
```

La evaluación de esta función calculará la llamada recursiva `(suma-hasta 4)`. Ahí es donde debemos **confiar en que la recursión hace bien su trabajo** y que esa llamada devuelve el valor resultante de 4+3+2+1, o sea, 10. Una vez obtenido ese valor hay que terminar el cálculo sumándole el propio número 5.

Otra característica necesaria del caso general en una definición recursiva, que también vemos en este ejemplo, es que **la llamada recursiva debe trabajar sobre un caso más sencillo que la llamada general**. De esta forma la recursión va descomponiendo el problema hasta llegar al caso base y construye la solución a partir de ahí.

En nuestro caso, la llamada recursiva para calcular la suma hasta 5 se hace calculando la suma hasta 4 (un caso más sencillo).

2.7.2. Diseño de la función `(suma-hasta x)`

¿Cómo hemos diseñado esta función? ¿Cómo hemos llegado a la solución?

Debemos empezar teniendo claro qué es lo que queremos calcular. Lo mejor es utilizar un ejemplo.

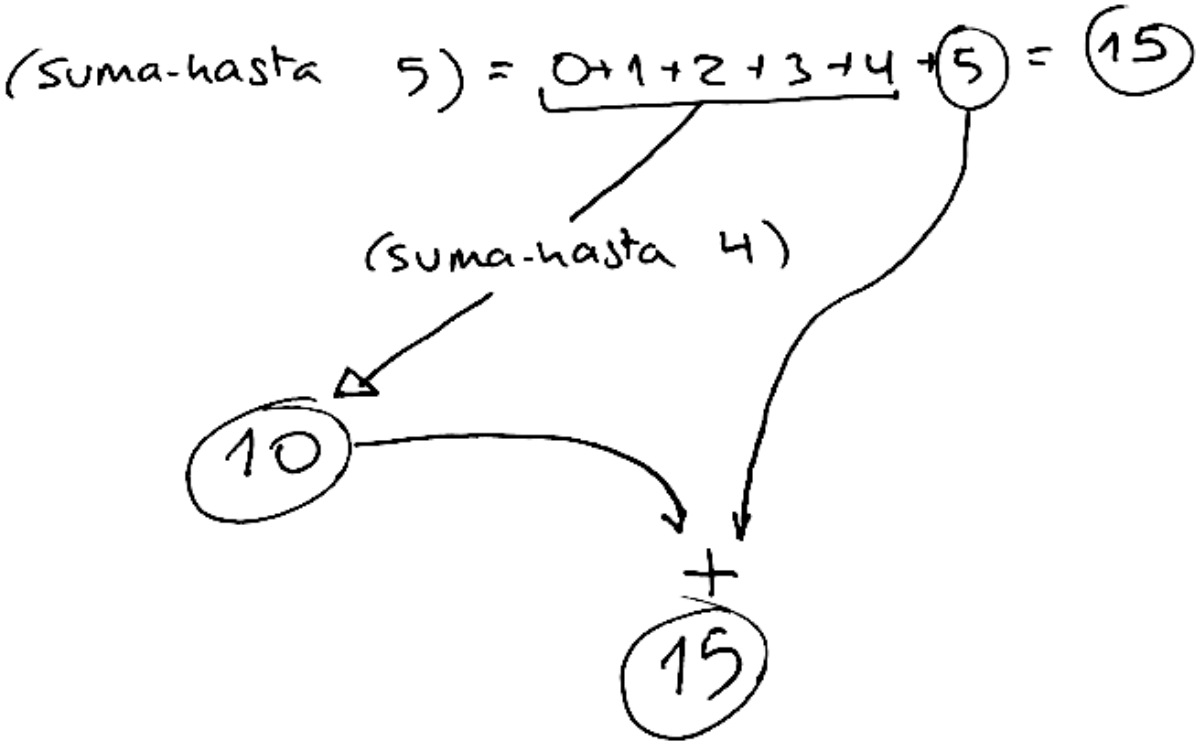
Por ejemplo, `(suma-hasta 5)` devolverá `0+1+2+3+4+5 = 15`.

Una vez que tenemos esta expresión de un ejemplo concreto debemos diseñar el caso general de la recursión. Para ello tenemos que encontrar una expresión para el cálculo de `(suma-hasta 5)` que **use una llamada recursiva** a un problema más pequeño.

O, lo que es lo mismo, ¿podemos obtener el resultado 15 con lo que nos devuelve una llamada recursiva que obtenga la suma hasta un número más pequeño y haciendo algo más?

Pues sí: para calcular la suma hasta 5, esto es, para obtener 15, podemos llamar a la recursión para calcular la suma hasta 4 (devuelve 10) y a este resultado sumarle el propio 5.

Lo podemos expresar con el siguiente dibujo:



Generalizamos este ejemplo y lo expresamos en Scheme de la siguiente forma:

```
(define (suma-hasta x)
   (+ (suma-hasta (- x 1)) x))
```

Nos falta el caso base de la recursión. Debemos preguntarnos **¿cuál es el caso más sencillo del problema, que podemos calcular sin hacer ninguna llamada recursiva?**. En este caso podría ser el caso en el que `x` es 0, en el que devolveríamos 0.

Podemos ya escribirlo todo en Scheme:

```
(define (suma-hasta x)
   (if (= 0 x)
      0
      (+ (suma-hasta (- x 1)) x)))
```

Una aclaración sobre el caso general. En la implementación anterior la llamada recursiva a `suma-hasta` se

realiza en el primer argumento de la suma:

```
(+ (suma-hasta (- x 1)) x)
```

La expresión anterior es totalmente equivalente a la siguiente en la que la llamada recursiva aparece como segundo argumento

```
(+ x (suma-hasta (- x 1)))
```

Ambas expresiones son equivalentes porque en programación funcional no importa el orden en el que se evalúan los argumentos. Da lo mismo evaluarlos de derecha a izquierda que de izquierda a derecha. La transparencia referencial garantiza que el resultado es el mismo.

2.7.3. Función `(alfabeto-hasta char)`

Vamos con otro ejemplo. Queremos diseñar una función `(alfabeto-hasta char)` que devuelva una cadena que empieza en la letra `a` y termina en el carácter que le pasamos como parámetro.

Por ejemplo:

```
(alfabeto-hasta #\h) ; ⇒ "abcdefgh"
(alfabeto-hasta #\z) ; ⇒ "abcdefghijklmnopqrstuvwxyz"
```

Pensamos en el caso general: ¿cómo podríamos invocar a la propia función `alfabeto-hasta` para que (confiando en la recursión) nos haga gran parte del trabajo (construya casi toda la cadena con el alfabeto)?

Podríamos hacer que la llamada recursiva devolviera el alfabeto hasta el carácter previo al que nos pasan como parámetro y después nosotros añadir ese carácter a la cadena que devuelve la recursión.

Veamos un ejemplo concreto:

```
(alfabeto-hasta #\h) = (alfabeto-hasta #\g) + \#h
```

La llamada recursiva `(alfabeto-hasta #\g)` devolvería la cadena `"abcdefg"` (confiando en la recursión) y sólo faltaría añadir la última letra.

Para implementar esta idea en Scheme lo único que necesitamos es usar la función `string-append` para concatenar cadenas y una función auxiliar `(anterior char)` que devuelve el carácter anterior a uno dado.

```
(define (anterior char)
  (integer->char (- (char->integer char) 1)))
```

El caso general quedaría como sigue:

```
(define (alfabeto-hasta char)
    (string-append (alfabeto-hasta (anterior char)) (string char)))
```

Faltaría el caso base. ¿Cuál es el caso más sencillo posible que nos pueden pedir? El caso del alfabeto hasta la `#\a`. En ese caso basta con devolver la cadena `"a"`.

La función completa quedaría así:

```
(define (alfabeto-hasta char)
  (if (equal? char #\a)
      "a"
      (string-append (alfabeto-hasta (anterior char)) (string char))))
```

2.8. Recursión y listas

La utilización de la recursión es muy útil para trabajar con estructuras secuenciales, como listas. Vamos a empezar viendo unos sencillos ejemplos y más adelante veremos algunos más complicadas.

2.8.1. Función recursiva `suma-lista`

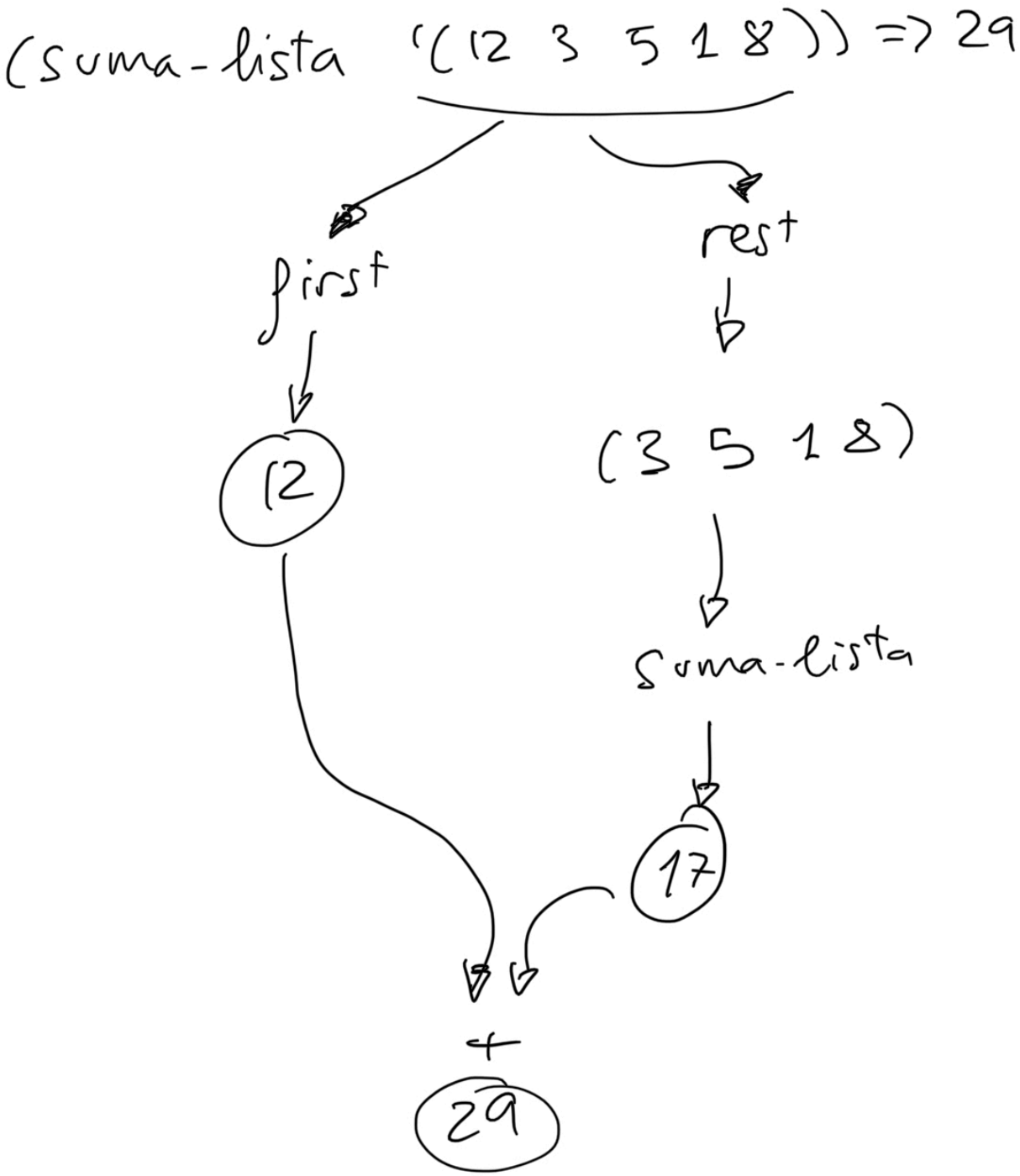
Veamos un primer ejemplo, la función `(suma-lista lista-nums)` que recibe como parámetro una lista de números y devuelve la suma de todos ellos.

Siempre debemos empezar escribiendo un ejemplo de la función, para entenderla bien:

```
(suma-lista '(12 3 5 1 8)) ; ⇒ 29
```

Para diseñar una implementación recursiva de la función tenemos que pensar en cómo descomponer el ejemplo en una llamada recursiva a un problema más pequeño y en cómo tratar el valor devuelto por la recursión para obtener el valor esperado.

Por ejemplo, en este caso podemos pensar que para sumar la lista de números `(12 3 5 1 8)` podemos obtener un problema más sencillo (una lista más pequeña) haciendo el `cdr` de la lista de números y llamando a la recursión con el resultado. La llamada recursiva devolverá la suma de esos números (confiamos en la recursión) y a ese valor basta con sumarle el primer número de la lista. Lo podemos representar en el siguiente dibujo:



Podemos generalizar este ejemplo y expresarlo en Scheme de la siguiente forma:

```
(define (suma-lista lista)
    (+ (first lista) (suma-lista (rest lista))))
```

Falta el caso base. ¿Cuál es la lista más sencilla con la que podemos calcular la suma de sus elementos sin llamar a la recursión?. Podría ser una lista sin elementos, y devolvemos 0. O una lista con un único elemento, y devolvemos el propio elemento. Escogemos como caso base el primero de ellos.

Con todo junto, la recursión quedaría como sigue:

```
(define (suma-lista lista)
```

```
    (if (null? lista)
        0
        (+ (first lista) (suma-lista (rest lista)))))
```

2.8.2. Función recursiva `longitud`

Veamos cómo definir la función recursiva que devuelve la longitud de una lista, el número de elementos que contiene.

Comencemos como siempre con un ejemplo:

```
(longitud '(a b c d e)) ; ⇒ 5
```

Suponiendo que la función `longitud` funciona correctamente, ¿cómo podríamos formular el caso general de la recursión? ¿cómo podríamos llamar a la recursión con un problema más pequeño y cómo podemos aprovechar el resultado de esta llamada para obtener el resultado final?

En este caso es bastante sencillo. Si a la lista le quitamos un elemento, cuando llamemos a la recursión nos va a devolver la longitud original menos uno. En este caso:

```
(longitud (rest '(a b c d e))) ; ⇒
(longitud '(b c d e )) ⇒ (confiamos en la recursión) 4
```

De esta forma, para conseguir la longitud de la lista inicial, sólo habría que sumarle 1 a lo que nos devuelve la llamada recursiva.

Si expresamos en Scheme este caso general:

```
; Sólo se define el caso general, falta el caso base
(define (longitud lista)
    (+ (longitud (rest lista)) 1))
```

Para definir el caso base debemos preguntarnos cuál es el caso más simple que le podemos pasar a la función. Si en cada llamada recursiva vamos reduciendo la longitud de la lista, el caso base recibirá la lista vacía. ¿Cuál es la longitud de una lista vacía? Una lista vacía no tiene elementos, por lo que es 0.

De esta forma completamos la definición de la función:

```
(define (longitud lista)
    (if (null? lista)
        0
        (+ (longitud (rest lista)) 1)))
```

En Scheme existe la función `length` que hace lo mismo. Devuelve la longitud de una lista:

```
(length '(a b c d e)) ; ⇒ 5
```

2.8.3. Cómo comprobar si una lista tiene un único elemento

En el caso base de algunas funciones recursivas es necesario comprobar que la lista que se pasa como

parámetro tiene un único elemento. Por ejemplo, en el caso base de la función recursiva que comprueba si una lista está ordenada.

Al estar definida la función `length` en Scheme la primera idea que se nos puede ocurrir es comprobar si la longitud de la lista es 1. Sin embargo es una mala idea.

```
; Ejemplo de función recursiva con un caso
; base en el que se comprueba si la lista tienen
; un único elemento
; ¡¡MALA IDEA, NO HACERLO ASÍ!!
(define (foo lista)
   (if (= (length lista) 1)
       ; devuelve caso base
       ; caso general
       ))
```

El problema de la implementación anterior es que el coste de la función `length` es lineal. Tal y como hemos visto en el apartado anterior, para calcular la longitud de la lista es necesario recorrer todos sus elementos. Además, la función recursiva hace esa comprobación en cada llamada recursiva. El coste resultante de la función `foo`, por tanto, es cuadrático.

¿Cómo mejorar el coste? Hay que tener en cuenta que la comprobación anterior está haciendo cosas de más. Realmente no queremos saber la longitud de la lista sino únicamente si esa longitud es mayor que uno. Esta comprobación sí que puede hacerse en tiempo constante. Lo único que debemos hacer es comprobar si el `cdr` de la lista es la lista vacía. Si lo es, ya sabemos que la lista original tenía un único elemento.

Por tanto, la versión correcto del código anterior sería la siguiente:

```
; Versión correcta para comprobar si una lista tiene
; un único elemento
(define (foo lista)
   (if (null? (rest lista))
       ; devuelve caso base
       ; caso general
       ))
```

El coste de la comprobación `(null? (rest lista))` es constante. No depende de la longitud de la lista.

2.8.4. Función recursiva `veces`

Como último ejemplo vamos a definir la función

```
(veces lista id)
```

que cuenta el número de veces que aparece un identificador en una lista.

Por ejemplo,

```
(veces '(a b c a d a) 'a ) ; ⇒ 3
```

¿Cómo planteamos el caso general? Llamaremos a la recursión con el resto de la lista. Esta llamada nos devolverá el número de veces que aparece el identificador en este resto de la lista. Y sumaremos al valor devuelto 1 si el primer elemento de la lista coincide con el identificador.

En Scheme hay que definir este caso general en una única expresión:

```
(if (equal? (first lista) id)
    (+ 1 (veces (rest lista) id))
    (veces (rest lista) id))
```

Como caso base, si la lista es vacía devolvemos 0.

La versión completa:

```
(define (veces lista id)
  (cond
    ((null? lista) 0)
    ((equal? (first lista) id) (+ 1 (veces (rest lista) id)))
    (else (veces (rest lista) id))))

(veces '(a b a a b b) 'a) ; ⇒ 3
```

3. Tipos de datos compuestos en Scheme

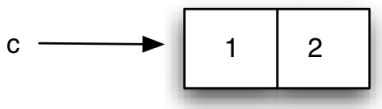
3.1. El tipo de dato pareja

3.1.1. Función de construcción de parejas `cons`

Ya hemos visto en el seminario de Scheme que el tipo de dato compuesto más simple es la pareja: una entidad formada por dos elementos. Se utiliza la función `cons` para construirla:

```
(cons 1 2) ; ⇒ (1 . 2)
(define c (cons 1 2))
```

Dibujamos la pareja anterior y la variable `c` que la referencia de la siguiente forma:



Tipo compuesto pareja

La instrucción `cons` construye un dato compuesto a partir de otros dos datos (que llamaremos izquierdo y derecho). La expresión `(1 . 2)` es la forma que el intérprete tiene de imprimir las parejas.

3.1.2. Construcción de parejas con `quote`

Al igual que las listas, es posible construir parejas con la forma especial `quote`, definiendo la pareja entre paréntesis y separando su parte izquierda y derecha con un punto:

```
'(1 . 2) ; ⇒ (1 . 2)
```

Utilizaremos a veces `cons` y otras veces `quote` para definir parejas. Pero hay que tener en cuenta que, al igual que con las listas, `quote` no evalúa sus parámetros, por lo que no lo deberemos utilizar por ejemplo dentro de una función en la que queremos construir una pareja con los resultados de evaluar expresiones.

Por ejemplo:

```
(define a 1)
(define b 2)
(cons a b) ; ⇒ (1 . 2)
'(a . b) ; ⇒ (a . b)
```

3.1.3. Funciones de acceso `car` y `cdr`

Una vez construida una pareja, podemos obtener el elemento correspondiente a su parte izquierda con la función `car` y su parte derecha con la función `cdr`:

```
(define c (cons 1 2))
(car c) ; ⇒ 1
(cdr c) ; ⇒ 2
```

3.1.3.1. DEFINICIÓN DECLARATIVA

Las funciones `cons`, `car` y `cdr` quedan perfectamente definidas con las siguientes ecuaciones algebraicas:

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

> **¿De dónde vienen los nombres `car` y `cdr`?**
>
> Inicialmente los nombres eran CAR y CDR (en mayúsculas). La historia se remonta al año 1959, en los orígenes del Lisp y tiene que ver con el nombre que se les daba a ciertos registros de la memoria del IBM 709.
>
> Podemos leer la explicación completa en The origin of CAR and CDR in LISP.

3.1.4. Función pair?

La función `pair?` nos dice si un objeto es atómico o es una pareja:

```
(pair? 3) ; ⇒ #f
(pair? (cons 3 4)) ; ⇒ #t
```

3.1.5. Las parejas pueden contener cualquier tipo de dato

Ya hemos comprobado que Scheme es un lenguaje *débilmente tipado*. Las funciones pueden devolver y recibir distintos tipos de datos.

Por ejemplo, podríamos definir la siguiente función `suma` que sume tanto números como cadenas:

```
(define (suma x y)
  (cond
    ((and (number? x) (number? y)) (+ x y))
    ((and (string? x) (string? y)) (string-append x y))
    (else 'error)))
```

En la función anterior los parámetros `x` e `y` pueden ser números o cadenas (o incluso de cualquier otro tipo). Y el valor devuelto por la función será un número, una cadena o el símbolo `'error`.

Sucede lo mismo con el contenido de las parejas. Es posible guardar en las parejas cualquier tipo de dato y combinar distintos tipos. Por ejemplo:

```
(define c (cons 'hola #f))
(car c) ; ⇒ 'hola
(cdr c) ; ⇒ #f
```

3.1.6. Las parejas son objetos inmutables

Recordemos que en los paradigmas de programación declarativa y funcional no existe el *estado mutable*. Una vez declarado un valor, no se puede modificar. Esto debe suceder también con las parejas: una vez creada una pareja no se puede modificar su contenido.

En Lisp y Scheme estándar las parejas sí que pueden ser mutadas. Pero durante toda esta primera parte de la asignatura no lo contemplaremos, para no salirnos del paradigma funcional.

En Swift y otros lenguajes de programación es posible definir **estructuras de datos inmutables** que no pueden ser modificadas una vez creadas. Lo veremos también más adelante.

3.2. Las parejas son objetos de primera clase

En un lenguaje de programación un elemento es de primera clase cuando puede:

- Asignarse a variables
- Pasarse como argumento
- Devolverse por una función
- Guardarse en una estructura de datos mayor

Las parejas son objetos de primera clase.

Una pareja puede asignarse a una variable:

```
(define p1 (cons 1 2))
(define p2 (cons #f "hola"))
```

Una pareja puede pasarse como argumento y devolverse en una función:

```
(define (suma-parejas p1 p2)
```

```
    (cons (+ (car p1) (car p2))
          (+ (cdr p1) (cdr p2))))

(suma-parejas '(1 . 5) '(4 . 12)) ; ⇒ (5 . 17)
```

Una vez definida esta función `suma-parejas` podríamos ampliar la función `suma` que vimos previamente con este nuevo tipo de datos:

```
(define (suma x y)
  (cond
    ((and (number? x) (number? y)) (+ x y))
    ((and (string? x) (string? y)) (string-append x y))
    ((and (pair? x) (pair? y)) (suma-parejas p1 p2))
    (else 'error)))
```

Y, por último, las parejas *pueden formar parte de otras parejas*.

Es lo que se denomina la **propiedad de clausura de la función `cons`**: el resultado de un `cons` puede usarse como parámetro de nuevas llamadas a `cons`.

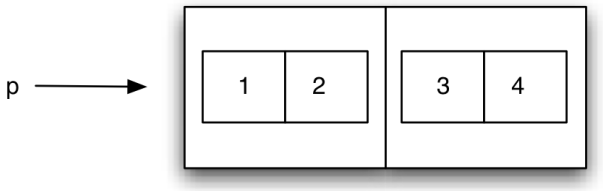
Ejemplo:

```
(define p1 (cons 1 2))
(define p2 (cons 3 4))
(define p (cons p1 p2))
```

Expresión equivalente:

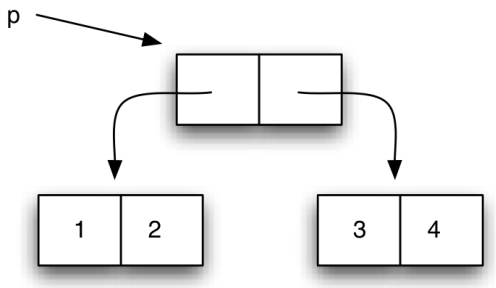
```
(define p (cons (cons 1 2)
                (cons 3 4)))
```

Podríamos representar esta estructura así:



Propiedad de clausura: las parejas pueden contener parejas

Pero se haría muy complicado representar muchos niveles de anidamiento. Por eso utilizamos la siguiente representación:



Llamamos a estos diagramas *diagramas caja-y-puntero* (*box-and-pointer* en inglés).

3.3. Diagramas *caja-y-puntero*

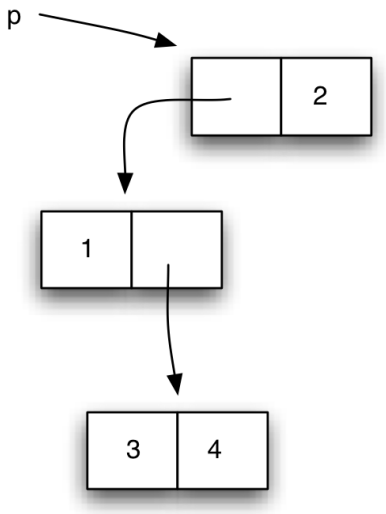
Al escribir expresiones complicadas con `cons` anidados es conveniente para mejorar su legibilidad utilizar el siguiente formato:

```
(define p (cons (cons 1
                      (cons 3 4))
                2))
```

Para entender la construcción de estas estructuras es importante recordar que las expresiones se evalúan *de dentro a afuera*.

¿Qué figura representaría la estructura anterior?

Solución:

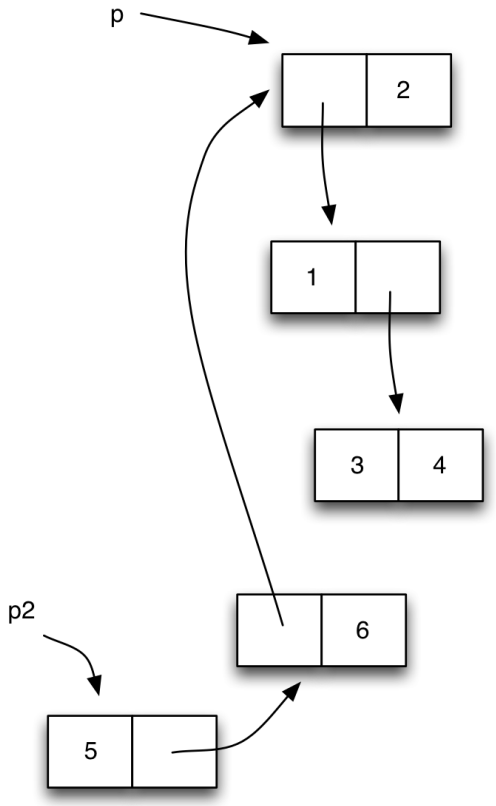


Es importante tener en cuenta que cada caja del diagrama representa una pareja creada en la memoria del intérprete con la instrucción `cons` y que el resultado de evaluar una variable en la que se ha guardado una pareja devuelve la pareja recién creada. Por ejemplo, si el intérprete evalúa `p` después de haber hecho la sentencia anterior devuelve la pareja contenida en `p`, no se crea una pareja nueva.

Por ejemplo, si después de haber evaluado la sentencia anterior evaluamos la siguiente:

```
(define p2 (cons 5 (cons p 6)))
```

El diagrama caja y puntero resultante sería el siguiente:



Vemos que en la pareja que se crea con `(cons p 6)` se guarda en la parte izquierda **la misma pareja que hay en `p`**. Lo representamos con una flecha que apunta a la misma pareja que `p`.

> **Nota**
>
> El funcionamiento de la evaluación de variables que contienen parejas es similar al de las variables que contienen objetos en lenguajes orientados a objetos como Java. Cuando se evalúa una variable que contiene una pareja se devuelve la propia pareja, no una copia.
>
> En programación funcional, como el contenido de las parejas es inmutable, no hay problemas de *efectos laterales* por el hecho de que una pareja esté compartida.

Es conveniente que pruebes a crear distintas estructuras de parejas con parejas y a dibujar su diagrama caja y puntero. Y también a recuperar un determinado dato (pareja o dato atómico) una vez creada la estructura.

La siguiente función `print-pareja` puede ser útil a la hora de mostrar por pantalla los elementos de una pareja

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

```
            (print-dato (cdr pareja))
            (display ")"))
        (display "")))

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

> **¡Cuidado!**
>
> La función anterior contiene pasos de ejecución con sentencias como `begin` y llamadas a `display` dentro del código de la función. Estas sentencias son propias de la programación imperativa. **No hacerlo en programación funcional**.

3.3.1. Funciones c????r

Al trabajar con estructuras de parejas anidades es muy habitual realizar llamadas del tipo:

```
(cdr (cdr (car p))) ; ⇒ 4
```

Es equivalente a la función `cadar` de Scheme:

```
(cddar p) ; ⇒ 4
```

El nombre de la función se obtiene concatenando a la letra "c", las letras "a" o "d" según hagamos un car o un cdr y terminando con la letra "r".

Hay definidas 2^4 funciones de este tipo: `caaaar`, `caaadr`, ..., `cddddr`.

4. Listas en Scheme

4.1. Implementación de listas en Scheme

Recordemos que Scheme permite manejar listas como un tipo de datos básico. Hemos visto funciones para crear, añadir y recorrer listas.

En Scheme las listas se implementan usando parejas, por lo que las funciones `car` y `cdr` también funcionan sobre listas.

¿Qué devuelven cuando se aplican a una lista? ¿Cómo se implementan las listas con parejas? Vamos a investigarlo haciendo unas pruebas.

En primer lugar, vamos a usar las funciones `list?` y `pair?` para comprobar si algo es una lista y/o una pareja.

Por ejemplo, una pareja formada por dos números es una pareja, pero no es una lista:

```
(define p1 (cons 1 2))
(pair? p1) ; ⇒ #t
(list? p1) ; ⇒ #f
```

Si preguntamos si una lista es una pareja, nos llevaremos la sorpresa de que sí. Una lista es una lista (evidentemente) pero también es una pareja:

```
(define lista '(1 2 3))
(list? lista); ⇒ #t
(pair? lista); ⇒ #t
```

Si una lista es también una pareja también podemos aplicar las funciones `car` y `cdr` con ellas. ¿Qué devuelven? Vamos a verlo:

```
(define lista '(1 2 3))
(car lista) ; ⇒  1
(cdr lista) ; ⇒  (2 3)
```

Resulta que en la pareja que representa la lista, en la parte izquierda se guarda el primer el elemento de la lista y en la parte derecha se guarda el resto de la lista.

También podemos explicar entonces por qué la llamada a `cons` con un dato y una lista construye otra lista:

```
(define lista '(1 2 3))
(define p1 (cons 1 lista))
(list? p1) ; ⇒  #t
p1 ; ⇒ (1 1 2 3)
```

¿Una pareja con una lista vacía como parte derecha es una lista? Lo probamos:

```
(define p1 (cons 1 '()))
(pair? p1) ; ⇒  #t
(list? p1) ; ⇒  #t
```

Con estos ejemplos ya tenemos pistas para deducir la relación entre listas y parejas en Scheme (y Lisp). Vamos a explicarlo.

4.1.1. Definición de listas con parejas

Una lista es:

- Una pareja que contiene en su parte izquierda el primer elemento de la lista y en su parte derecha el resto de la lista
- Un símbolo especial `'()` que denota la lista vacía

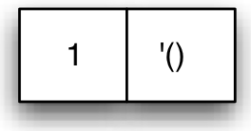
Hay que notar que la definición anterior es una definición recursiva.

Por ejemplo, una lista muy sencilla con un solo elemento, `(1)`, se define con la siguiente pareja:

```
(cons 1 '())
```

La pareja cumple las condiciones anteriores:

- La parte izquierda de la pareja es el primer elemento de la lista (el número 1)
- La parte derecha es el resto de la lista (la lista vacía)



La lista (1)

El objeto es al mismo tiempo una pareja y una lista. La función `list?` permite comprobar si un objeto es una lista:

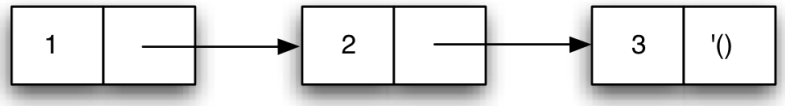
```
(define l (cons 1 '()))
(pair? l)
(list? l)
```

Por ejemplo, la lista '(1 2 3) se construye con la siguiente secuencia de parejas:

```
(cons 1
      (cons 2
            (cons 3
                  '())))
```

La primera pareja cumple las condiciones de ser una lista:

- Su primer elemento es el 1
- Su parte derecha es la lista '(2 3)



Parejas formando una lista

Al comprobar la implementación de las listas en Scheme, entendemos por qué las funciones `car` y `cdr` nos devuelven el primer elemento y el resto de la lista. De hecho, las funciones `first` y `rest` se implementan usando las funciones `car` y `cdr`.

4.1.2. Lista vacía

La lista vacía es una lista:

```
(list? '()) ; ⇒ #t
```

Y no es un símbolo ni una pareja:

```
(symbol? '()) ; ⇒ #f
(pair? '()) ; ⇒ #f
```

Para saber si un objeto es la lista vacía, podemos utilizar la función `null?`:

```
(null? '()) ; ⇒ #t
```

En Racket está predefinido el símbolo `null` que tiene como valor la lista vacía:

```
null ; ⇒ ()
```

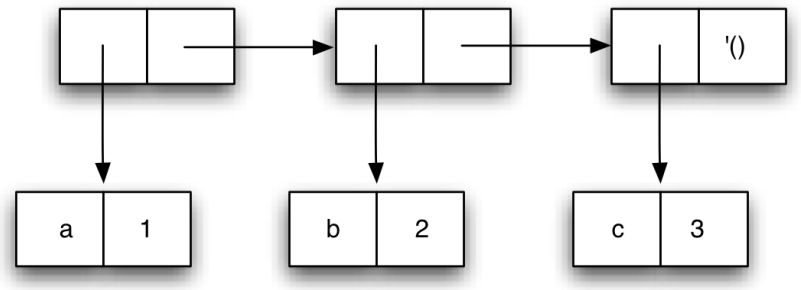
4.2. Listas con elementos compuestos

Las listas pueden contener cualquier tipo de elementos, incluyendo otras parejas.

La siguiente estructura se denomina *lista de asociación*. Son listas cuyos elementos son parejas (*clave*, *valor*):

```
(list (cons 'a 1)
      (cons 'b 2)
      (cons 'c 3)) ; ⇒ ((a . 1) (b . 2) (c . 2))
```

¿Cuál sería el diagrama *box and pointer* de la estructura anterior?



La expresión equivalente utilizando conses es:

```
(cons (cons 'a 1)
      (cons (cons 'b 2)
            (cons (cons 'c 3)
                  '())))
```

4.2.1. Listas de listas

Hemos visto que podemos construir listas que contienen otras listas:

```
(define lista (list 1 (list 1 2 3) 3))
```

La lista anterior también se puede definir con quote:

```
(define lista '(1 (1 2 3) 3))
```

La lista resultante contiene tres elementos: el primero y el último son elementos atómicos (números) y el segundo es otra lista.

Si preguntamos por la longitud de la lista Scheme nos dirá que es una lista de 3 elementos:

```
(length lista) ; ⇒ 3
```

Y el segundo elemento de la lista es otra lista:

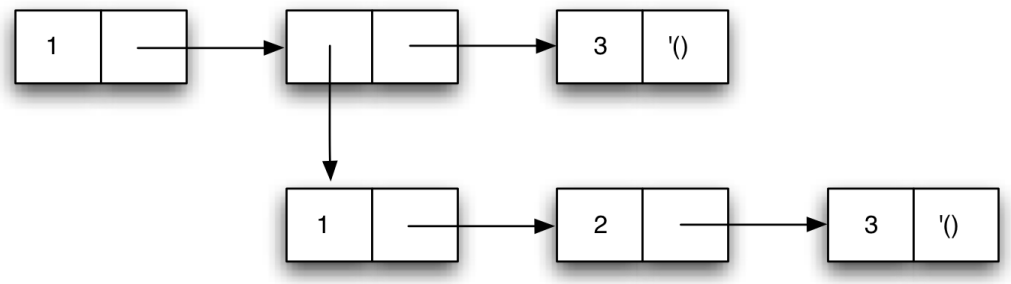
```
(second lista) ; ⇒ (1 2 3)
```

> **Funciones second, third, ..., tenth**
>
> En Racket existen funciones que devuelven el segundo, tercer, ... y así hasta el décimo elemento de una lista. Son las funciones `second`, `third`, ..., `tenth`. Se pueden consultar en el manual de referencia del lenguaje.

¿Cómo implementa Scheme esta lista usando parejas?

Al ser una lista de tres elementos lo hará con tres parejas enlazadas que terminan en una lista vacía en la parte derecha de la última pareja. En las partes izquierdas de esas tres parejas tendremos los elementos de la lista propiamente dichos: un 1 y un 3 en la primera y última pareja y una lista en la segunda pareja.

El diagrama *box and pointer*:



Lista que contiene otra lista como segundo elemento

4.2.2. Impresión de listas y parejas por el intérprete de Scheme

El intérprete de Scheme siempre intenta mostrar una lista cuando encuentra una pareja cuyo siguiente elemento es otra pareja.

Por ejemplo, si tenemos la siguiente estructura:

```
(define p (cons 1 (cons 2 3)))
```

Cuando se evalúe `p` el intérprete imprimirá por pantalla lo siguiente:

```
(1 2 . 3)
```

¿Por qué? Porque el intérprete va construyendo la salida conforme recorre la pareja `p`. Como encuentra una pareja cuya parte derecha es otra pareja, lo interpreta como el comienzo de una lista, y por eso escribe `(1 2` en lugar de `(1 . 2`. Pero inmediatamente después se encuentra con el `3` en lugar de una lista vacía. En ese momento el intérprete "se da cuenta" de que no tenemos una lista y termina la expresión escribiendo el `. 3` y el paréntesis final.

Si queremos comprobar la estructura de parejas podemos utilizar la función `print-pareja` definida anteriormente, que imprimiría lo siguiente:

```
(print-pareja p) ; ⇒ (1 . (2 . 3))
```

4.2.3. Funciones de alto nivel sobre listas

Es importante conocer cómo se implementan las listas usando parejas y su representación con diagramas caja y puntero para definir funciones de alto nivel.

Una vez conocidos los detalles de implementación, podemos volver a usar las funciones que tienen un nivel de abstracción mayor como `first` y `rest`. Son funciones que tienen un nombre entendible y que comunican perfectamente lo que hacen (devolver el primer elemento y el resto).

```
(first '(a b c d)) ; ⇒ a
(rest '(a b c d)) ; ⇒ (b c d)
```

Existen otras funciones de alto nivel que trabajan sobre listas. Algunas ya las conocemos, pero otras no:

```
(append '(a (b) c) '((d) e f)) ; ⇒ (a (b) c (d) e f)
(list-ref '(a (b) c d) 2) ; ⇒ c
(length '(a (b (c))) ; ⇒ 2
(reverse '(a b c))  ; ⇒ (c b a)
(list-tail '(a b c d) 2) ; ⇒ (c d)
```

En los siguientes apartados veremos cómo están implementadas.

4.3. Funciones recursivas que construyen listas

Para terminar el apartado sobre las listas en Scheme vamos a ver ejemplos adicionales de funciones recursivas que trabajan con listas. Veremos alguna función que recibe una lista y, como antes, usa la recursión para recorrerla. Pero veremos también funciones que usan la recursión para **construir nuevas listas**.

Algunas de las funciones que presentamos son implementaciones de las ya existentes en Scheme. Para no solapar con las definiciones de Scheme pondremos el prefijo `mi-` en todas ellas.

Vamos a ver las siguientes funciones:

- `mi-list-ref`: implementación de la función `list-ref`

- `mi-list-tail`: implementación de la función `list-tail`
- `mi-append`: implementación de la funión `append`
- `mi-reverse`: implementación de la función `reverse`
- `cuadrados-hasta`: devuelve la lista de cuadrados hasta uno dado
- `filtra-pares`: devuelve la lista de los números pares de la lista que se recibe
- `primo?`: comprueba si un número es o no primo

4.3.1. Función `mi-list-ref`

La función `(mi-list-ref n lista)` devuelve el elemento `n` de una lista (empezando a contar por 0):

```
(define lista '(a b c d e f g))
(mi-list-ref lista 2) ; ⇒ c
```

Veamos con el ejemplo anterior cómo hacer la formulación recursiva.

Hemos visto que, en general, cuando queremos resolver un problema de forma recursiva tenemos que hacer una llamada recursiva a un problema más sencillo, **confiar en que la llamada nos devuelva el resultado correcto** y usar ese resultado para resolver el problema original.

En este caso nuestro problema es obtener el número que está en la posición 2 de la lista `(a b c d e f g)`. Suponemos que la función que nos devuelve una posición de la lista ya la tenemos implementada y que la llamada recursiva nos va a devolver el resultado correcto. ¿Cómo podemos simplificar el problema original? Veamos la solución para este caso concreto:

```
Para devolver el elemento 2 de la lista (a b c d e f g):
   Obtenemos el resto de la lista (b c d e f g)
   y devolvemos su elemento 1. Será el valor c (empezamos
   a contar por 0).
```

Generalizamos el ejemplo anterior, para cualquier `n` y cualquier lista:

```
Para devolver el elemento que está en la posición `n` de una lista,
devuelvo el elemento n-1 de su resto.
```

Y, por último, formulamos el caso base de la recursión, el problema más sencillo que se puede resolver directamente, sin hacer una llamada recursiva:

```
Para devolver el elemento que está en la posición 0 de una lista,
devuelvo el `first` de la lista.
```

La implementación de todo esto en Scheme sería la siguiente:

```
(define (mi-list-ref lista n)
   (if (= n 0)
      (first lista)
      (mi-list-ref (rest lista) (- n 1))))
```

4.3.2. Función `mi-list-tail`

La función `(mi-list-tail lista n)` devuelve la lista resultante de quitar n elementos de la cabeza de la lista original:

```
(mi-list-tail '(1 2 3 4 5 6 7) 2) ; ⇒ (3 4 5 6 7)
```

Piensa en cómo se implementaría de forma recursiva. Esta vez vamos a mostrar directamente la implementación, sin dar explicaciones de cómo se ha llegado a ella:

```
(define (mi-list-tail lista n)
   (if (= n 0)
       lista
       (mi-list-tail (rest lista) (- n 1))))
```

4.3.3. Función `mi-append`

Veamos ahora cómo podríamos implementar de forma recursiva la función `append` que une dos listas. La llamaremos `(mi-append lista1 lista2)`.

Por ejemplo:

```
(mi-append '(a b c) '(d e f)) ; ⇒ (a b c d e f)
```

Para resolver el problema de forma recursiva, debemos confiar en la recursión para que resuelva un problema más sencillo y después terminar de arreglar el resultado devuelto por la recursión.

En este caso, podemos pasarle a la recursión un problema más sencillo quitando el primer elemento de la primera lista (con la función `rest`) y llamando a la recursión para que concatene esta lista más pequeña con la segunda. Confiamos en que la recursión funciona correctamente y nos devuelve la concatenación de ambas listas

```
(mi-append (rest '(a b c)) '(d e f)) => (b c d e f)
```

Y añadiremos el primer elemento a la lista resultante usando un `cons`:

```
(mi-append '(a b c) '(d e f)) =
(cons 'a (mi-append '(b c)) '(d e f)) =
(cons 'a '(b c d e f)) =
(a b c d e f)
```

En general:

```
(define (mi-append lista1 lista2)
   (cons (first lista1) (mi-append (rest lista1) lista2)))
```

El caso base, el caso en el que la función puede devolver un valor directamente sin llamar a la recursión, es aquel en el que `lista1` es `null?`. En ese caso devolvemos `lista2`:

```
(mi-append '() '(a b c)) => '(a b c)
```

La formulación recursiva completa queda como sigue:

```
(define (mi-append l1 l2)
    (if (null? l1)
        l2
        (cons (first l1)
              (mi-append (rest l1) l2))))
```

4.3.4. Función `mi-reverse`

Veamos cómo implementar de forma recursiva la función `mi-reverse` que invierte una lista

```
(mi-reverse '(1 2 3 4 5 6)) ; ⇒ (6 5 4 3 2 1)
```

La idea es sencilla: llamamos a la recursión para hacer la inversa del `rest` de la lista y añadimos el primer elemento a la lista resultante que devuelve ya invertida la llamada recursiva.

Podemos definir una función auxiliar `(añade-al-final dato lista)` que añade un dato al final de una lista usando `append`:

Veamos directamente su implementación, usando `mi-append` para añadir un elemento al final de la lista:

```
(define (añade-al-final dato lista)
    (append lista (list dato)))
```

La función `mi-reverse` quedaría entonces como sigue:

```
(define (mi-reverse lista)
    (if (null? lista) '()
    (añade-al-final (first lista) (mi-reverse (rest lista)))))
```

4.3.5. Función `cuadrados-hasta`

La función `(cuadrados-hasta x)` devuelve una lista con los cuadrados de los números hasta `x`:

```
Para construir una lista de los cuadrados hasta x:
   construyo la lista de los cuadrados hasta x-1 y le añado el cuadrado de x
```

El caso base de la recursión es el caso en el que `x` es 1, entonces devolvemos una lista formada por el 1.

En Scheme:

```
(define (cuadrados-hasta x)
   (if (= x 1)
      '(1)
      (cons (cuadrado x)
            (cuadrados-hasta (- x 1)))))
```

Ejemplo:

```
(cuadrados-hasta 10) ; ⇒ (100 81 64 49 36 25 16 9 4 1)
```

4.3.6. Función `filtra-pares`

Es muy habitual recorrer una lista y comprobar condiciones de sus elementos, construyendo una lista con los que cumplan una determinada condición.

Por ejemplo, la siguiente función `filtra-pares` construye una lista con los números pares de la lista que le pasamos como parámetro:

```
(define (filtra-pares lista)
   (cond
      ((null? lista) '())
      ((even? (first lista))
       (cons (first lista)
          (filtra-pares (rest lista))))
      (else (filtra-pares (rest lista)))))
```

En el caso general, llamamos de forma recursiva a la función para que filtre el `rest` de la lista. Y le añadimos el primer elemento si es par.

Cada vez llamaremos a la recursión con una lista más pequeña, por lo que en el caso base tendremos que comprobar si la lista que recibimos. En ese caso devolvemos la lista vacía.

Ejemplo:

```
(filtra-pares '(1 2 3 4 5 6)) ; ⇒ (2 4 6)
```

4.3.7. Función `primo?`

El uso de listas es uno de los elementos fundamentales de la programación funcional.

Como ejemplo, vamos a ver cómo trabajar con listas para construir una función que calcula si un número es primo. La forma de hacerlo será calcular la lista de divisores del número y comprobar si su longitud es dos. En ese caso será primo.

Por ejemplo:

```
(divisores 8) ; ⇒ (1 2 4 8) longitud = 4, no primo
(divisores 9) ; ⇒ (1 3 9) longitud = 3, no primo
(divisores 11) ; ⇒ (1 11) longitud = 2, primo
```

Podemos definir entonces la función `(primo? x)` de la siguiente forma:

```
(define (primo? x)
   (=  2
      (length (divisores x))))
```

¿Cómo implementamos la función `(divisores x)` que nos devuelve la lista de los divisores de un número `x`. Vamos a construirla de la siguiente forma:

1. Creamos una lista de todos los números del 1 a x
2. Filtramos la lista para dejar los divisores de x

La función `(lista-desde x)` devuelve una lista de números x..1:

```
(define (lista-desde x)
   (if (= x 0)
      '()
      (cons x (lista-desde (- x 1)))))
```

Ejemplos:

```
(lista-desde 2) ; ⇒ (2 1)
(lista-desde 10) ; ⇒ (10 9 8 7 6 5 4 3 2 1)
```

Definimos la función `(divisor? x y)` que nos diga si x es divisor de y:

```
(define (divisor? x y)
      (= 0 (mod y x)))
```

Ejemplos:

```
(divisor 2 10) ; ⇒ #t
(divisor 3 10) ; ⇒ #f
```

Una vez que hemos definido La función `divisor?` podemos utilizarla para definir la función recursiva `(filtra-divisores lista x)` que devuelve una lista con los números de `lista` que son divisores de `x`:

```
(define (filtra-divisores lista x)
   (cond
      ((null? lista) '())
      ((divisor? (first lista) x)
       (cons (first lista)
             (filtra-divisores (rest lista) x)))
      (else (filtra-divisores (rest lista) x))))
```

Ya podemos implementar la función que devuelve los divisores de un número `x` generando los números hasta `x` y filtrando los divisores de ese número. Por ejemplo, para calcular los divisores de 10:

```
(filtra-divisores (1 2 3 4 5 6 7 8 9 10) 10) ; ⇒ (1 2 5 10)
```

Se puede implementar de una forma muy sencilla:

```
(define (divisores x)
   (filtra-divisores (lista-disde x) x))
```

Y una vez definida esta función, ya puede funcionar correctamente la función `primo?`.

4.4. Funciones con número variable de argumentos

Hemos visto algunas funciones primitivas de Scheme, como `+` o `max` que admiten un número variable de argumentos. ¿Podemos hacerlo también en funciones definidas por nosotros?

La respuesta es sí, utilizando lo que se denomina notación *dotted-tail* (punto-cola) para definir los parámetros de la función. En esta notación se coloca un punto antes del último parámetro. Los parámetros antes del punto (si existen) tendrán como valores los argumentos usados en la llamada y el resto de argumentos se pasarán en forma de lista en el último parámetro.

Por ejemplo, si tenemos la definición

```
(define (funcion-dos-o-mas-args x y . lista-args)
    <cuerpo>)
```

podemos llamar a la función anterior con dos o más argumentos:

```
(funcion-dos-o-mas-args 1 2 3 4 5 6)
```

En la llamada, los parámetros `x` e `y` tomarán los valores 1 y 2. El parámetro `lista-args` tomará como valor una lista con los argumentos restantes `(3 4 5 6)`.

También es posible permitir que todos los argumentos sean opcionales no poniendo ningún argumento antes del punto::

```
(define (funcion-cualquier-numero-args . lista-args)
    <cuerpo>)
```

Si hacemos la llamada

```
(funcion-cualquier-numero-args 1 2 3 4 5 6)
```

el parámetro `lista-args` tomará como valor la lista `(1 2 3 4 5 6)`.

Veamos un sencillo ejemplo.

Podemos implementar una función `mi-suma` que tome al menos dos argumentos y después un número variable de argumentos y devuelva la suma de todos ellos. Es muy sencillo: recogemos todos los argumentos en la lista de argumentos variables y llamamos a la función `suma-lista` que suma una lista de números:

```
(define (mi-suma x y . lista-nums)
    (if (null? lista-nums)
        (+ x y)
        (+ x (+ y (suma-lista lista-nums)))))
```

5. Funciones como tipos de datos de primera clase

Hemos visto que la característica fundamental de la programación funcional es la definición de funciones. Hemos visto también que no producen efectos laterales y no tienen estado. Una función toma unos datos como entrada y produce un resultado como salida.

Una de las características fundamentales de la programación funcional es considerar a las funciones como *objetos de primera clase*. Recordemos que un tipo de primera clase es aquel que:

1. Puede ser asignado a una variable
2. Puede ser pasado como argumento a una función
3. Puede ser devuelto como resultado de una invocación a una función
4. Puede ser parte de un tipo mayor

Vamos a ver que las funciones son ejemplos de todos los casos anteriores: vamos a poder crear funciones sin nombre y asignarlas a variables, pasarlas como parámetro de otras funciones, devolverlas como resultado de invocar a otra función y guardarlas en tipos de datos compuestos como listas.

La posibilidad de usar funciones como objetos de primera clase es una característica fundamental de los lenguajes funcionales. Es una característica de muchos lenguajes multi-paradigma con características funcionales como JavaScript, Python, Swift o a partir de la versión 8 de Java, Java 8, (donde se denominan *expresiones lambda*).

5.1. Forma especial `lambda`

Vamos a empezar explicando la forma especial `lambda` de Scheme, que nos permite crear funciones anónimas en tiempo de ejecución.

De la misma forma que podemos usar cadenas o enteros sin darles un nombre, en Scheme es posible usar una función sin darle un nombre mediante esta forma especial.

5.1.1. Sintaxis de la forma especial `lambda`

La sintaxis de la forma especial `lambda` es:

```
(lambda (<arg1> ... <argn>)
    <cuerpo>)
```

El cuerpo del lambda define un *bloque de código* y sus argumentos son los parámetros necesarios para ejecutar ese bloque de código. Llamamos a la función resultante una *función anónima*.

Algunos ejemplos:

Una función anónima que suma dos parejas:

```
(lambda (p1 p2)
    (cons (+ (car p1) (car p2))
          (+ (cdr p1) (cdr p2))))
```

Una función anónima que devuelve el mayor de dos números:

```
(lambda (a b)
    (if (> a b)
        a
        b))
```

5.1.2. Semántica de la forma especial `lambda`

La invocación a la forma especial `lambda` construye una función anónima en tiempo de ejecución.

Por ejemplo, si ejecutamos una expresión lambda en el intérprete veremos que devuelve un procedimiento:

```
(lambda (x) (* x x)) ; ⇒ #<procedure>
```

El procedimiento construido es un bloque de código que devuelve el cuadrado de un número.

¿Qué podemos hacer con este procedimiento?

Podemos asignarlo a un identificador. Por ejemplo, en la siguiente expresión, primero se evalúa la *expresión lambda* y el procedimiento resultante se asocia al identificador `f`.

```
(define f (lambda (x) (* x x)))
```

El ejemplo anterior funciona de una forma idéntica al siguiente:

```
(define x (+ 2 3))
```

En ambos casos se evalúa la expresión derecha y el resultado se guarda en un identificador. En el primer caso la expresión que se evalúa devuelve un procedimiento, que se guarda en la variable `f` y en el segundo un número, que se guarda en la variable `x`.

Si escribimos los identificadores `f` y `x` en el intérprete Scheme los evalúa y muestra los valores guardados:

```
f ; ⇒ #<procedure:f>
x ; ⇒ 5
```

En el primer caso se devuelve un procedimiento y en el segundo un número. Fíjate que Scheme trata a los procedimientos y a los números de la misma forma; son lo que se denominan datos de primera clase.

Una vez asignado un procedimiento a un identificador, lo podemos utilizar como de la misma forma que invocamos habitualmente a una función:

```
(f 3) ; ⇒ 9
```

No es necesario un identificador para invocar a una función; podemos crear la función con una expresión lambda e invocar a la función anónima recién creada:

```
((lambda (x) (* x x)) 3) ; ⇒ 9
```

La llamada a `lambda` crea un procedimiento y el paréntesis a su izquierda lo invoca con el parámetro 3:

```
((lambda (x) (* x x)) 3) ; ⇒ (#<procedure> 3) ⇒ 9
```

Es importante remarcar que con `lambda` estamos creando una función en *tiempo de ejecución*. Es código que creamos para su posterior invocación.

Cada lenguaje de programación tiene su sintaxis propia de expresiones lambda. Por ejemplo, las siguientes expresiones crean una función que devuelve el cuadrado de un número:

Java 8

```
Integer x -> {x*x}
```

Scala

```
(x:Int) => {x*x}
```

Objective C

```
^int (int x)
{
   x*x
};
```

Swift

```
{ (x: Int) -> Int in return x*x }
```

5.1.3. Identificadores y funciones

Tras conocer `lambda` ya podemos explicarnos por qué cuando escribimos en el intérprete de Scheme el nombre de cualquier función, se evalúa a un *procedure*:

```
+ ; ⇒ <procedure:+>
append ; ⇒ #<procedure:append>
```

El identificador se evalúa y devuelve el *objeto función* al que está ligado. En Scheme los nombres de las funciones son realmente símbolos a los que están ligados *objetos de tipo función*.

Podemos comprobar también de esta manera que `and` y `or` no son funciones. Si escribimos `and` o `or` e intentamos evaluar cualquiera de los dos símbolos, veremos que Scheme devuelve un error:

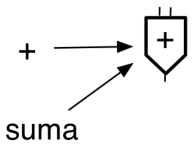
```
and
; and: bad syntax in: and
or
; or: bad syntax in: or
```

Podemos asignar funciones ya existentes a nuevos identificadores usando `define`, como en el ejemplo

siguiente:

```
+ ; ⇒ <procedure:+>
(define suma +)
(suma 1 2 3 4) ; ⇒ 10
```

Es muy importante darse cuenta que la expresión `(define suma +)` se evalúa de forma idéntica a `(define y x)`. Primero se evalúa el identificador `+`, que devuelve el *objeto función* suma, que se asigna a la variable `suma`. El resultado final es que tanto `+` como `suma` tienen como valor el mismo procedimiento:



La forma especial `define` para definir una función no es más que *azucar sintáctico*.

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

siempre se convierte internamente en:

```
(define <nombre>
    (lambda (<args>)
        <cuerpo>))
```

Por ejemplo

```
(define (cuadrado x)
    (* x x))
```

es equivalente a:

```
(define cuadrado
    (lambda (x) (* x x)))
```

5.1.4. Predicado `procedure?`

Podemos comprobar si algo es una función utilizando el predicado de Scheme `procedure?`.

Por ejemplo:

```
(procedure? (lambda (x) (* x x))) ; ⇒ #t
(define suma +)
(procedure? suma) ; ⇒ #t
(procedure? '+) ; ⇒ #f
```

Hemos visto que las funciones pueden asignarse a variables. También cumplen las otras condiciones necesarias para ser consideradas objetos de primera clase.

5.2. Funciones argumentos de otras funciones

Hemos visto ya un ejemplo de cómo pasar una función como parámetro de otra. Veamos algún otro.

Por ejemplo, podemos definir la función `aplica` que recibe una función en el parámetro `func` y dos valores en los parámetros `x` e `y` y devuelve el resultado de invocar a la función que pasamos como parámetro con `x` e `y`. La función que se pase como parámetro debe tener dos argumentos

Para realizar la invocación a la función que se pasa como parámetro basta con usar `func` como su nombre. La función se ha ligado al nombre `func` en el momento de la invocación a `aplica`, de la misma forma que los argumentos se ligan a los parámetros `x` e `y`:

```
(define (aplica f x y)
   (f x y))
```

Algunos ejemplos de invocación, usando funciones primitivas, funciones definidas y expresiones lambda:

```
(aplica + 2 3) ; ⇒ 5
(aplica * 4 5) ; ⇒ 10
(aplica string-append "hola" "adios") ; ⇒ "holaadios"

(define (string-append-con-guion s1 s2)
    (string-append s1 "-" s2))

(aplica string-append-con-guion "hola" "adios") ; ⇒ "hola-adios"

(aplica (lambda (x y) (sqrt (+ (* x x) (* y y)))) 3 4) ; ⇒ 5
```

Otro ejemplo, la función `aplica-2` que toma dos funciones `f` y `g` y un argumento `x` y devuelve el resultado de aplicar `f` a lo que devuelve la invocación de `g` con `x`:

```
(define (aplica-2 f g x)
   (f (g x)))

(define (suma-5 x)
   (+ x 5))
(define (doble x)
   (+ x x))
(aplica-2 suma-5 doble 3) ; ⇒ 11
```

5.3. Función `apply`

La función `(apply funcion lista)` de Scheme permite aplicar una función de aridad `n` a una lista de datos de n datos, haciendo que cada uno de los datos se pasen a la función en orden como parámetros.

La función `apply` recibe una función y una lista y devuelve el resultado de aplicar la función a los datos de la lista, tomándolos como parámetros.

Por ejemplo, podemos aplicar la función suma a una lista de números:

```
(apply + '(1 2 3 4)) ; ⇒ 10
```

Podemos pasar a `apply` una expresión lambda:

```
(apply (lambda (x y) (+ x (* 2 y))) '(2 5)) ; ⇒ 12
```

La lista que pasamos como argumento de `apply` debe tener tantos elementos como parámetros tenga la función que aplicamos. En caso contrario, se produce un error:

```
(apply cons '(a b c)) ; ⇒ error
cons: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 2
  given: 3
  arguments...:
```

La forma correcta de hacerlo:

```
(apply cons '(a b)) ; ⇒ (a . b)
```

5.3.1. Función `apply` y funciones recursivas

Usando `apply` podemos definir funciones recursivas con número variable de argumentos.

Por ejemplo, supongamos que queremos definir la función `suma-parejas` que suma un número variable de parejas:

```
(suma-parejas '(1 . 2) '(3 . 4) '(5 . 6)) ; ⇒ '(9 . 12)
```

Recordemos la definición de la función que suma dos parejas:

```
(define (suma-pareja p1 p2)
  (cons (+ (car p1) (car p2))
        (+ (cdr p1) (cdr p2))))
```

Podemos entonces construir la función `suma-parejas` que recibe una lista de parejas (número variable de argumentos) llamando a `apply` para que sume todas las parejas del resto de la lista. Y sumar la pareja resultante con la primera pareja de la lista:

```
(define (suma-parejas . parejas)
  (if (null? parejas)
      '(0 . 0)
      (suma-pareja (first parejas) (apply suma-parejas (rest parejas)))))
```

Se trata de una llamada recursiva indirecta, porque se invoca a la propia función `suma-parejas` no directamente, sino a través de `apply`.

Hay que hacer notar en que la llamada recursiva es necesario usar `apply` porque `(cdr parejas)` es una lista. No podemos invocar a `suma-parejas` pasando una lista como parámetro, sino que hay que pasarle todos los argumentos por separado (recibe un número variable de argumentos). Eso lo conseguimos hacer con `apply`.

La siguiente imagen muestra una representación gráfica del funcionamiento de la recursión: