Tema 3: Programación funcional

Contenidos

Bibliografía

- SICP: Cap. 1.1.1–1.1.6, 1.3, 2.2 (2.2.1) 2.3.1
- PLP: Cap. 10
- Concepts in Programming Languages Cap. 4.4

Objetivos

Semana 1

- Conocer los hitos más importantes en la historia del paradigma funcional
- Conocer las características más importantes del paradigma funcional
- Saber diferenciar entre lenguajes y sentencias imperativas y lenguajes y sentencias declarativas
- Aplicar el modelo de computación de sustitución (en orden aplicativo y normal) para la evaluación de expresiones
- Diferenciar entre funciones y formas especiales en Scheme
- Conocer y saber utilizar las formas especialies principales de Scheme
- Conocer el tipo *símbolo* de Scheme y las funciones que trabajan con ellos
- Conocer y saber utilizar la forma especial `eval` y la dualidad entre datos y programas
- Comprender el uso de la recursión como forma de implementar la iteración en

Scheme y lenguajes funcionales
- Comprender funciones recursivas sencillas que trabajen sobre estructuras compuestas (listas y cadenas)

1. El paradigma de Programación Funcional

1.1. Definición e historia del paradigma funcional

En un sentido estricto, la programación funcional define un programa como una función matemática que convierte unas entradas en unas salidas, sin ningún estado interno y ningún efecto lateral.

La no existencia de estado interno (celdas de memoria en las que se guardan y se modifican valores, por ejemplo) y la ausencia de efectos laterales es una característica también de la **programación declarativa**.

Otras características del paradigma funcional son las siguientes:

- Recursión
- Funciones como tipos de datos primitivos
- Uso de listas

1.1.1 Orígenes históricos

- Cálculo lambda (Alonzo Church) en los años 30
- Años 30 distintos matemáticos proponen modelos computacionales y demuestran su equivalencia (Turing - Máquina de Turing, Kleen - Sustituciones algebráicas, …)
- Modelo de cálculo lambda basado en la definición de funciones y la aplicación de estas funciones a argumentos

1.1.2 Historia del Lisp

- Lisp es el primer lenguaje de programación de alto nivel basado en el paradigma funcional
- Creado en 1958 por John McCarthy
- Lisp es un lenguaje revolucionario e introduce nuevos conceptos de programación: funciones como objetos primitivos, funciones de orden superior, polimorfismo, listas, recursión, símbolos, homogeneidad de datos y programas, bucle REPL “read-eval-print”
- 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

1.1.3. Lisp no es un lenguaje estricto de programación funcional

En Lisp (y en Scheme) existen instrucciones que se salen del paradigma funcional puro y

permiten estado local y efectos laterales (programación imperativa). Por ello son lenguajes en los que es posible programar de forma imperativa, no funcional. Nosotros no vamos a utilizar esas instrucciones en esta primera parte de la asignatura, escribiendo siempre código 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.

Con el paso de los años y el avance en los diseños de compiladores e intérpretes ha sido posible diseñar lenguajes de programación que siguen más estrictamente las características declarativas del paradigma funcional y que también son útiles y prácticos para desarrollar programas en el mundo real, como Haskell, Miranda o ML.

1.2. Programación declarativa

Hablamos de *programación declarativa* para referirnos a lenguajes de programación (o sentencias de código) en los que se *declaran* los valores, objetivos o características finales de los elementos del programa, pero no se especifican detalles de implementación, ni de control de flujo. Estos elementos se resuelven por la componente de run-time del lenguaje.

Por ejemplo, un conjunto de reglas de Prolog son sentencias declarativas. O una definición de una interfaz en Java.

Una característica fundamental del código declarativo es que no utiliza pasos de ejecución, ni asignación destructiva. 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 que son declarativos (como el Prolog). De la misma forma que existen lenguajes que tienen características funcionales y que no son declarativos (como el Lisp o Scheme).

En esta primera parte de la asignatura, en la que vamos a tratar el paradigma de programación funcional, vamos a usar sólo las características declarativas de Scheme.

Avanzamos 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)
- 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
- No existen referencias

Características de la programación imperativa

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

1.2.1. Programación declarativa vs. imperativa

El estilo de programación imperativa se basa en pasos de ejecución que modifican el estado de variables:

```
int x = x + 1;
int y = y + 3;
```

Las expresiones anteriores son típicas expresiones de asignación que modifican valores anteriores de una variable por nuevos valores. El *estado* de las variables (su valor) cambia con la ejecución de los pasos del programa.

Por contra, un ejemplo de programación declarativa en el que *declaramos* una función que toma como entrada un número y devuelve su cuadrado:

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

Otra característica de la programación declarativa es que no existe el estado local mutable.

En programación imperativa esto no es así. Por ejemplo, una de las características de la programación orientada a objetos es guardar estado mutable en variables de instancia de clases.

Por ejemplo, en Java:

```
public class Contador {
        int c;

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

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

```
}

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

Cada llamada al método `valor()` modifica el estado del objeto `cont`.

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

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

        c++;
        return c;
}

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

1.2.2. Las funciones devuelven siempre el mismo valor

Los lenguajes funcionales puros tienen la propiedad de *transparencia referencial*: es posible sustituir una expresión por su valor sin que se introduzca ninguna modificación en el resultado final del programa.

Como consecuencia, 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.

1.2.3. Diferencia entre declaración y modificación de variables

En programación funcional pura una vez declarada una variable no se puede modificar su valor. En algunos lenguajes de programación (como Scala) este concepto se refuerza definiendo la variable como inmutable (con la directiva `val`).

En programación imperativa es habitual modificar el valor de una variable en distintos pasos de ejecución

Ejemplo:

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

Líneas 1 y 3: sentencias declarativas
Líneas 2, 4: sentencias imperativas

1.3. Valores y referencias

En programación declarativa sólo existen valores, no hay referencias. Cuando se realiza una asignación **de un valor** a una variable debemos considerar que estamos dando un nombre a un objeto matemático que no puede ser modificado o que estamos copiando el valor en la variable.

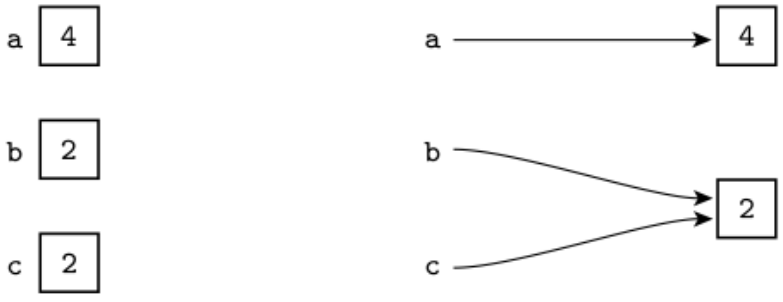
Por ejemplo, en Java, los tipos de datos primitivos son valores. Las asignaciones valores de estos tipos a variables realizan copias de valores:

```
int a = 4;
int b = 2;
int c = b;
b = 3;
```

En la variable `a` se copia el valor 4 y en las variables `b` y `c` se copia el valor `2`. No hay forma de modificar (*mutar*) esos valores. Podríamos cambiar las variables guardando en ella otros valores, pero los valores propiamente dichos son inmutables. En la última instrucción modificamos el valor de la variable `b`, pero el valor de la variable `c` sigue siendo 2.

Los tipos de datos cuyos valores son inmutables y sus asignaciones tienen una semántica de copia reciben el nombre de **tipos de valor** (*value types* en inglés).

En programación imperativa, sin embargo, existe la distinción entre valores y referencias.



Diferencia entre valor y referencia

Las variables `b` y `c` son referencias que están apuntando a una misma posición, en la que se encuentra el número 2. Si modificamos (*mutamos*) el contenido de esa posición se modifica también el valor de las dos variables.

Este comportamiento es el de los **tipos de referencia**. Se trata de tipos de datos mutables en los que la asignación funciona con semántica de referencia.

Distintos lenguajes de programación tienen distintas formas de trabajar con los tipos de referencia. Por ejemplo, cualquier objeto en Java tiene una semántica de referencia. Cuando asignamos un objeto a una variable, estamos guardando en la variable una referencia al objeto.

```
Point2D p1 = new Point2D(3.0, 2.0); // 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 mod
```

Es fundamental entender la semántica de referencia, porque distintas variables pueden referenciar un mismo objeto y se pueden producir **efectos laterales** (*side effects* en inglés). El dato guardado en una variable cambia después de una sentencia en la que no se ha usado esa variable.

1.3.1 Asignación

En programación funcional no existe la *asignación destructiva*, en la que se modifica un valor previamente asignado. Sí que se pueden dar valor a variables o identificadores, entendiéndolo como una definición de un valor constante.

La forma especial `define` en Scheme crea una nueva variable y le da el valor definido.

```
(define mi-nombre "Alejandro Perez")
```

Sin embargo, en programación imperativa una variable guarda una referencia a una posición de memoria (o estructura de datos) que puede ser modificada posteriormente mediante una nueva asignación:

```
char *miNombre = "Alejandro Perez"
miNombre[3] = 'a'
```

En programación funcional debemos entender las variables como variables matemática que tienen un valor y que tienen la propiedad de transparencia referencial. No debemos entenderlas como una referencia a una posición de memoria que puede ser modificada. Los valores son inmutables y no existen efectos laterales.

En esta página de la Wikipedia se explica bastante bien la semántica de la asignación en programación funcional.

1.3.2. Igualdad en lenguajes con referencias

En los lenguajes que trabajan con referencias es necesario definir dos tipos de igualdades: igualdad de valor e igualdad de referencia.

Por ejemplo, en Java se utiliza la igualdad `==` para comprobar la igualdad de referencia

y el método `equals` para la igualdad de valor:

```
String s1 = "Hola";
String s2 = s1;
String s3 = "Hola";
Boolean v1 = (s1 == s2); // true
Boolean v2 = (s1 == s3); // false
Boolean v3 = (s1.equals(s3)); // true
```

1.3.3. Ventajas e inconvenientes de las referencias mutables

El uso de referencias es común en la mayoría de los lenguajes modernos. En lenguajes como Python, Ruby, Smalltalk, Objective-C, etc. la asignación funciona como en Java, copiando las referencias a los objetos. Esto permite construir grafos de objetos relacionados en tiempo de ejecución que mantienen el estado compartido de la aplicación.

El estado compartido mutable es muy eficiente, porque con una única modificación de un único objeto se actualiza todas las variables que están apuntando a él. Pero el estado compartido mutable también puede hacer que el código sea difícil de mantener y que sea complicado razonar matemáticamente sobre su corrección.

El código que se añade al proyecto puede modificar el comportamiento del código ya existente, pudiendo introducirse bugs al aumentar las funcionalidades del programa.

El código escrito en programación funcional no contiene referencias mutables y está libre de los efectos laterales. La construcción de un programa se puede hacer de forma incremental, añadiendo funciones que nunca van a modificar el comportamiento de las funciones ya existentes.

Scheme también tiene sentencias de mutación, pero no las usaremos en esta primera parte de la asignatura en la que estamos escribiendo código funcional.

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 va a ser el resultado de evaluar una determinada expresión.

El modelo de computación de sustitución es un modelo muy sencillo que permite definir la semántica de la evaluación de expresiones en lenguajes funcionales como Scheme.

Es muy fácil de implementar: como reto proponemos que lo escribáis en forma de programa en algún lenguaje que conozcáis (Java, Python, …)

El modelo de sustitución se basa en una versión simplificada de la regla de reducción del

cálculo lambda.

Reglas para evaluar una expresión *e* usando el modelo de sustitución:

1. Si *e* es un valor primitivo, devolver ese mismo valor.
2. Si *e* es una variable, devolver su valor asociado con un `define`.
3. Si *e* es una expresión del tipo *(f arg1 … argn)*, donde *f* el nombre de una función primitiva ('+', '-', …), evaluar *arg1* … *argn* y llamar a la función con el resultado.
4. Si *e* es una expresión del tipo *(f arg1 … argn)*, donde *f* el nombre de una función definida por el usuario (definida con un `define` en Scheme), sustituir *f* por su cuerpo, reemplazando cada parámetro formal del procedimiento por el correspondiente argumento evaluado. Evaluar la expresión resultante.

1.4.1 Orden normal vs. orden aplicativo

En el orden aplicativo se realizan las evaluaciones 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.

Scheme utiliza el orden aplicativo.

1.4.2. Ejemplo

Supongamos las siguientes definiciones de funciones:

```
(define (double x) (+ x x))
(define (square y) (* y y))
(define (f z) (+ (square (double z)) 1))
```

¿Cuál sería el resultado de evaluar `(f (+ 2 1))` con orden aplicativo?

Solución:

```
(f (+ 2 1)) ->              ; evaluamos (+ 2 1) por ser + una función primitiva
(f 3) ->                    ; sustituimos (f 3) por su definición
(+ (square (double 3)) 1) -> ; sustituimos (double 3)
(+ (square (+ 3 3)) 1) ->    ; evaluamos (+ 3 3)
(+ (square 6) 1) ->          ; sustituimos (square 6)
(+ (* 6 6) 1) ->             ; evaluamos (* 6 6)
(+ 36 1) ->                  ; evaluamos (+ 36 1)
37
```

¿Y en orden normal?

```
(f (+ 2 1)) ->                    ; sustituimos (f (+ 2 1))
```

```
                                    ; por su definición, con z = (+ 2 1)
(+ (square (double (+ 2 1))) 1) -> ; sustituimos (double (+ 2 1))
(+ (square (+ (+ 2 1)
              (+ 2 1))) 1) ->      ; sustituimos (square ...)
(+ (* (+ (+ 2 1)
         (+ 2 1))
      (+ (+ 2 1)
         (+ 2 1))) 1) ->           ; evaluamos (+ 2 1)
(+ (* (+ 3 3)
      (+ 3 3)) 1) ->               ; evaluamos (+ 3 3)
(+ (* 6 6) 1) ->                   ; evaluamos (* 6 6)
(+ 36 1) ->                        ; evaluamos (+ 36 1)
37
```

En programación funcional el resultado de evaluar una expresión es el mismo independientemente del tipo de orden.

1.4.3. El orden de evaluación sí importa si no tenemos programación funcional

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))
```

1.5. Aplicaciones prácticas de la programación funcional

1.5.1. El renacimiento de la programación funcional

En los años 60 la programación funcional (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 *multi-paradigma* (muchos de ellos interpretados) como Ruby, Python, Groovy, Objective-C, Lua o Scala que incluyen el paradigma funcional.

La gran ventaja del paradigma funcional es la ausencia de estado. Estos lenguajes aprovechan esta característica para implementar programas fácilmente escalables en arquitecturas de múltiples procesadores concurrentes o de múltiples servidores.

Un 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:

> "Scala encourages no shared state. Finagle is assumed correct because it's tested by Twitter in production. Mutable state is avoided using constructs in Scala or Finagle. No long running state machines are used. State is pulled from the database, used, and written back to the database. Advantage is developers don't need to worry about threads or locks."

Otro ejemplo es el uso de las técnicas de programación funcional en los nuevos sistemas de bases de datos NoSQL como es el *MapReduce* de Google. Como dice Joel Spolsky en su artículo The Perils of JavaSchools:

> "Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable."

1.5.2. La programación funcional refuerza la metodología de *programación evolutiva*

Los programas complejos se construyen a base de ir definiendo y probando elementos computacionales cada vez más complicados.

Abelson y Sussman comentan en el SICP:

> In general, computational objects may have very complex structures, and it would be extremely inconvenient to have to remember and repeat their details each time we want to use them. Indeed, *complex programs are constructed by building, step by step, computational objects of increasing complexity*.
>
> The interpreter makes this step-by-step program construction particularly convenient because name-object associations can be created *incrementally in successive interactions*. This feature encourages the *incremental development and testing of programs* and is largely responsible for the fact that a Lisp program usually consists of a large number of relatively simple procedures

Esta metodología de programación se denomina programación iterativa o evolutiva.

2. Scheme como lenguaje de programación funcional

Vamos ver ejemplos concretos de características de un lenguaje de programación funcional estudiando las características funcionales de Scheme.

En concreto, veremos:

- Definición de funciones y otras primitivas de programación funcional en Scheme
- Símbolos y primitivas `quote` y `eval`

- 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: define, if, cond

2.2.1 Forma especial `define`

Sintaxis

```
(define <símbolo> <expresión>)
```

Evaluación

1. Evaluar <*expresión*>
2. Asociar el valor resultante con el <*símbolo*>

Ejemplo

```
(define base 10)
(define altura 12)
(define area (/ (* base altura) 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
                (factorial (- x 1))))
```

2.2.2. 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))
```

2.2.3. 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 expresiones hasta que una de ellas devuelva `#t`
2. Si alguna expresión devuelve `#t`, se devuelve el valor del consecuente de esa expresión
3. Si ninguna expresión es cierta, se devuelve el valor resultante de evaluar el consecuente del `else`

Ejemplo

```
(cond
   ((> 3 4) '3-es-mayor-que-4)
   ((< 2 1) '2-es-menor-que-1)
   ((= 3 1) '3-es-igual-que-1)
   ((= 2 2) '2-es-igual-que-2)
   ((> 3 2) '3-es-mayor-que-2)
   (else 'ninguna-condicion-es-cierta))
```

2.3. Símbolos y formas especiales `quote` y `eval`

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**.

Los símbolos son distintos de las cadenas. Una cadena es un tipo de dato compuesto, mientras que los símbolos se almacenan con un valor único denominado *valor hash*.

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) ; #t
(symbol? "hola") ; #f
(symbol?  #f) ; #f
(symbol? (car '(hola cómo estás))) ; #t
(equal? 'hola 'hola)
(equal? 'hola "hola")
```

2.3.1. Forma especial `quote`

Sintaxis

```
(quote <simbolo>)
(quote <expresion>)
```

Evaluación

Se devuelve el símbolo o la expresión sin evaluar. Las expresiones deben estar definidas con un número balanceado de paréntesis y definen listas de elementos que pueden ser recorridas con las funciones `car` y `cdr`. Se abrevia en con el carácter `'`

Ejemplo

```
(quote x)
'hola
'(+ 1 2 3 4)
(quote (1 2 3 4))
'(* (+ 1 (+ 2 3)) 5)
```

2.3.2. Evaluación de símbolos

Un símbolo es un identificador que puede asociarse o ligarse (*bind*) a un valor (cualquier dato *de primera clase*).

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

```
(define pi 3.14159)
pi
⇒3.14159
```

Los nombres de las funciones (`equal?,` sin, `+, …) son también símbolos (los de las macros no) 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))
⇒ procedure:cuadrado
```

2.3.4. Símbolos como tipos primitivos

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

```
(define x 'hola)
x
⇒ hola
```

2.3.5. La forma especial `eval`

La forma especial `eval` es una característica importantísima de los lenguajes funcionales (incluyendo Scheme) que no existe en la mayoría de lenguajes imperativos.

Permite evaluar la expresión que le pasamos como parámetro:

Sintaxis

```
(eval <expresión>)
```

Evaluación

Se evalúa la <*expresión*> y se devuelve el resultado.

Ejemplos

- Para que funcionen los siguientes ejemplos hay que cambiar el lenguaje del DrRacket a *Pretty Big*, en R5RS `eval` necesita argumentos adicionales.

Probamos los siguientes ejemplos para entender mejor el funcionamiento de símbolos y valores:

```
(eval 8)
(define a 8)
(define b 'a)
a
b
(eval 'a)
(eval b)
(eval 'b)
(eval '(+ 1 2 3))
```

2.3.6. Dualidad entre datos y programas

La forma especial `eval` permite una característica importante de la programación funcional: la dualidad entre datos y programas.

Los programas en Scheme son expresiones entre paréntesis y una expresión es una lista de símbolos. Esto permite tratar a los programas como datos y viceversa.

Por ejemplo:

```
(define mi-lista (list '+ 1 2 3 4))
(eval mi-lista)
```

Otro ejemplo: supongamos que queremos sumar una lista de números

```
(define (suma-lista lista-nums)
(eval (cons '+ lista-nums)))
```

2.3.7. Un intérprete de Scheme en Scheme

Históricamente, la introducción de `eval` en Lisp fue el momento en el que se descubrió la posibilidad de hacerlo interpretado.

A continuación vemos un ejemplo de intérprete de Scheme utilizando la forma especial `eval`. Es una implementación muy sencilla del bucle "Read-Eval-Print" del intérprete de Scheme. Utiliza alguna forma especial que no hemos visto todavía como `let`, para definir un nuevo ámbito en el que se declara una nueva variable (en este caso, la variable `expr` en la que se guarda la expresión que el usuario introduce que se lee con la función `read`).

Para que funcione el código en el intérprete *DrRacket* hay que seleccionar el lenguaje "Muy grande" o "Pretty Big".

¡Cuidado!: el ejemplo utiliza características de programación imperativa como los pasos de ejecución o la forma especial `begin`.

```
(define (rep-loop)
   (display "mi-interprete> ")  ; imprime un prompt
   (let ((expr (read)))            ; lee una expresión
      (if (eq? expr 'adios)        ; el usuario quiere parar?
         (begin
            (display "saliendo del bucle read-eval-print")
            (newline))
         (begin                   ; expresión distinta de 'adios
            (write (eval expr))   ; evaluar e imprimir
            (newline)
            (rep-loop)))))
```

2.4. Listas

Otra de las características fundamentales del paradigma funcional es la utilización de listas. Repasamos las características y funciones más importantes de Scheme para trabajar con listas.

- Creación de listas: función `list` y forma especial `quote`

  ```
  (define a 1)
  (define b 2)
  (define c 3)
  (define lista2 (list a b c))
  (define lista1 '(a b c))
  ```

- Lista vacía: `()`

- Lista con otras listas como elemento:

  ```
  (define lista4 '(1 (2 3) (4 5 (6))))
  ```

  La `lista4` tiene 3 elementos:

  - 1

- La lista '(2 3)
- La lista '(4 5 (6))

- Selección de elementos de una lista:
  - Primer elemento: función `car`
  - Resto de elementos: función `cdr`

```
(define lista3 '((1 2) 3 4))
(car lista3) ;-> (1 2)
(cdr lista3) ;-> (3 4)
```

- Creación de nuevas listas:

  - Función `cons` para crear una lista nueva resultado 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)
```

  - Función `append` para crear una lista nueva resultado de concatenar dos o más listas

```
(append list1 list2 list3)
```

2.5. 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

Las iteraciones se realizan con recursión.

Para entender correctamente la recursión hay que mirarla *de forma declarativa*, como una definición matemática, entendiendo lo que hace la llamada recursiva y *confiando en que devuelve lo que tiene que devolver*. No es conveniente *entrar en la recursión* e intentar comprobar su funcionamiento haciendo una traza de las sucesivas llamadas.

> Confía en la recursión.

2.5.1. Función factorial

Ejemplo típico, factorial:

```
(define (factorial x)
   (if (= x 0)
```

```
      1
      (* x (factorial (- x 1)))))
```

Una interpretación declarativa (*definición matemática*) de la llamda recursiva sería:

> Para calcular el factorial de *x* debemos multiplicar *x* por el factorial de *x–1*.

Lo cual es cierto, dada la siguiente relación:

factorial x = x * (x–1) * (x–2) * … * 1 = x * factorial (x–1)

2.5.2. Función `suma-hasta`

Función `(suma-hasta x)` que suma los números 0+1+2+…+x.

Definición matemática:

> Para sumar desde 0 hasta *x* debemos sumar a *x* el resultado de sumar desde 0 hasta *x–1*.

En Scheme:

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

2.5.3. Función `longitud`

Podemos calcular la longitud de una lista con la siguiente expresión recursiva:

> La longitud de una lista *l* es 1 + la longitud del resto de *l*

En Scheme, la función `(longitud lista)` que calcula la longitud de una lista:

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

2.5.4. Función `suma-lista`

La suma de todos los números de una lista de enteros se puede calcular con la siguiente expresión recursiva:

> La suma de los números de una lista es el primer elemento + la suma de los elementos del resto.

En Scheme:

```
(define (suma-lista lista)
   (if (null? lista)
       0
           (+ (car lista) (suma-lista (cdr lista)))))
```

2.5.5. Función recursiva `veces`

Vamos a definir la función `(veces pal car)` que cuenta el número de veces que aparece un carácter en una palabra (una cadena).

Definimos las funciones auxiliares `primero` y `resto`:

```
(define (primero pal)
   (string-ref pal 0))

(define (resto pal)
   (substring pal 1 (string-length pal)))
```

La función `veces` se define recursivamente de la siguiente forma:

> Para calcular el número de veces que aparece una letra *c* en una palabra, cuento el número de veces que aparece en el resto de la palabra y le sumo 1 si la primera letra coincide con *c*.

En Scheme:

```
(define (veces pal car)
   (cond
      ((equal? "" pal) 0)
      ((equal? (primero pal) car) (+ 1 (veces (resto pal) car))
      (else (veces (resto pal) car)))))
```

2.5.6. Función recursiva `codifica`

Vamos a definir la función `(codifica cadena)` que transforma una cadena por los siguientes caracteres.

Definimos las funciones `siguiente` y `anterior` sobre caracteres

```
(define (siguiente car)
   (integer->char (+ 1 (char->integer car))))

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

Y la función `codifica` se define como sigue:

> Para codificar una palabra concatena la codificación del primer carácter con el resto

> de la palabra codificada.

En Scheme:

```
(define (codifica pal)
   (if (equal? "" pal)
      ""
      (string-append (string (siguiente (primero pal)))
                     (codifica (resto pal)))))
```

Ejemplo:

```
(codifica "hola que tal")
⇒ "ipmb!rvf!ubm"
```

La función `descodifica` hace lo contrario

```
(define (descodifica pal)
   (if (equal? "" pal)
      ""
      (string-append (string (anterior (primero pal)))
                     (descodifica (resto pal)))))
```

Ejemplo:

```
(descodifica "ipmb!rvf!ubm")
⇒ "hola que tal"
```

2.5.7. Ejemplo de uso de la recursión para construir listas

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, añado el cuadrado de x a la lista de los cuadrados hasta x–1

Es muy importante el caso base: si x=1, devuelvo 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)
```

2.5.8. Ejemplo de recursión que construye una lista nueva

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? (car lista)) (cons (car lista)
                                     (filtra-pares (cdr lista))))
      (else (filtra-pares (cdr lista)))))
```

Ejemplo:

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

2.5.9. Función `primo?`

Terminamos con un ejemplo completo en el que vamos a construir de forma incremental la función `primo?` que comprueba si un número es primo. En principio no nos va a preocupar la eficiencia; consideradlo sólo un ejemplo de una solución recursiva basada en listas.

Un número es primo cuando sólo tiene dos divisores, el número 1 y él mismo. Definimos entonces la función `(primo? x)` de la siguiente forma:

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

Suponemos que la función `(divisores x)` nos devuelve una lista con todos los divisores del 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-hasta x)` devuelve una lista de números 1..x:

```
(define (lista-hasta x)
   (if (= x 0)
```

```
      '()
      (cons x (lista-hasta (- x 1)))))
```

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

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

La función `(filtra-divisores lista x)` devuelve una lista con los números de la lista original que son divisores de x

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

Por último, la función `(divisores x)` devuelve la lista con los divisores de un número:

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

Os propongo como ejercicio que encontréis una solución más eficiente.
