Tema 1: Lenguajes de programación

Índice

1. Lenguajes de programación

1.1. Elementos de los lenguajes de programación

Según la definición de la *Encyclopedia of Computer Science* (Encyclopedia of Computer Science, 4th Edition, Anthony Ralston (Editor), Edwin D. Reilly (Editor), David Hemmendinger (Editor), Wiley, 2000. Disponible en la biblioteca politécnica con identificador: POE R0/E/I/ENC/RAL):

*"A **programming language** is a set of characters, rules for combining them, and rules specifying their effects when executed by a computer, which have the following four characteristics:*

1. *It requires no knowledge of machine code on the part of the user*
2. *It has machine independence*
3. *Is translated into machine language*
4. *Employs a notation that is closer to that of the specific problem being solved than is machine code"*

Según Abelson y Sussman, en el libro de texto de nuestra asignatura (SICP, p. 1):

*"We are about to study the idea of a **computational process**. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called **data**. The evolution of a process is directed by a pattern of rules called a **program**. [...] The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric programming languages that prescribe the tasks we want our processes to perform."*

Y después, en la página 4, añaden otra idea fundamental:

"A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. Thus, when we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas."

Así, entre las características de un lenguaje de programación podemos remarcar las siguientes:

- Define un proceso que se ejecuta en un computador
- Es de alto nivel, cercano a los problemas que se quieren resolver (abstracción)
- Permite construir nuevas abstracciones que se adapten al dominio que se programa

Para Abelson y Sussman, todos los lenguajes de progamación permiten combinar ideas

simples en ideas más complejas mediante los siguientes tres mecanismos:

- **expresiones primitivas**, que representan las entidades más simples del lenguaje
- **mecanismos de combinación** con los que se construyen elementos compuestos a partir de elementos más simples
- **mecanismos de abstracción** con los que dar nombre a los elementos compuestos y manipularlos como unidades

Cuando se habla de *elementos* en el párrafo anterior nos estamos refiriendo tanto a datos como a programas.

1.2. Historia de los lenguajes de programación

Desde 1954 hasta la actualidad se han documentado más de 2.500 lenguajes de programación (consultar en The Language List (http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm)). Entre 1952 y 1972, la primera época de los lenguajes de programación, se desarrollaron alrededor de 200 lenguajes, de los que una decena fueron realmente siginificativos y tuvieron influencia en el desarrollo de lenguajes posteriores.

Una lista parcial de algunos de los lenguajes de programación más importantes, junto con su año de creación:

- 1957 FORTRAN
- 1958 ALGOL
- 1960 Lisp
- 1960 COBOL
- 1962 APL
- 1962 SIMULA
- 1964 BASIC
- 1964 PL/I
- 1970 Prolog
- 1972 C
- 1975 Pascal
- 1975 Scheme
- 1975 Modula
- 1983 Smalltalk-80
- 1983 Objective-C
- 1983 Ada
- 1986 C++
- 1986 Eiffel
- 1987 Perl

- 1988 Tcl/Tk
- 1990 Haskell
- 1991 Python
- 1993 Ruby
- 1995 Java
- 1995 PHP
- 2000 C#

Éric Lévénez ha hecho el esfuerzo de construir un árbol genealógico de los lenguajes de programación, que indica la fecha en la que cada lenguaje fue creado y la influencia que ha tenido en los lenguajes posteriores. Consultar el documento PDF (docs/HistoryProgrammingLanguagesLevenez_a4.pdf) y la web de Éric Lévénez (http://www.levenez.com/lang/) .

2. Paradigmas de programación

El origen de la palabra *paradigma* entendida como un marco general en el que se desarrollan teorías científicas se encuentra en el trabajo de 1962 del filósofo e historiador de la ciencia Thomas S. Kuhn, *La estructura de las revoluciones científicas*. Esa palabra ha sido después adoptada por el mundo de la computación para definir un conjunto de ideas y principios comunes de grandes grupos de lenguajes de programación.

La definición de la palabra *paradigma* más cercana a lo que se quiere decir en la expresión *paradigma de programación* es la siguiente:

"*Un marco filosófico y teórico de una escuela o disciplina científica en el que se formulan teorías, leyes y generalizaciones y los experimentos realizados en soporte de ellas*."

Un paradigma define un conjunto de reglas, patrones y estilos de programación que son usados por los lenguajes de programación que usan ese paradigma.

Podemos distinguir cuatro grandes paradigmas de programación:

- Paradigma funcional
- Paradigma lógico
- Paradigma imperativo o procedural
- Paradigma orientado a objetos

Algunas características importantes de cada uno de estos paradigmas.

2.1. Paradigma funcional

En el paradigma funcional la computación se realiza mediante la evaluación de expresiones.

- Definición de funciones
- Funciones como datos primitivos
- Valores sin efectos laterales, no existe la asignación
- Programación declarativa

2.2. Paradigma lógico

- Definición de reglas
- Unificación como elemento de computación
- Programación declarativa

2.3. Paradigma imperativo

Los lenguajes de programación que complen el paradigma imperativo se caracterizan por tener un estado implícito que es modificado mediante instrucciones o comandos del lenguaje. Como resultado, estos lenguajes tienen una noción de secuenciación de los comandos para permitir un control preciso y determinista del estado.

- Definición de procedimientos
- Definición de tipos de datos
- Chequeo de tipos en tiempo de compilación
- Cambio de estado de variables
- Pasos de ejecución de un proceso

2.4. Paradigma orientado a objetos

- Definición de clases y herencia
- Objetos como abstracción de datos y procedimientos
- Polimorfismo y chequeo de tipos en tiempo de ejecución

Una reflexión importante es que la separación entre los paradigmas y los lenguajes no es estricta. Existen ideas comunes a distintos paradigmas, así como lenguajes de programación que soportan más de un paradigma. Por ejemplo, el paradigma funcional y lógico comparten características *declarativas*, mientras que el paradigma orientado a objetos y procedural tienen características *imperativas*.

Otros paradigmas de programación menos comunes:

- Paradigmas de programación paralela y concurrente
- Paradigmas basados en restricciones
- Paradigmas visuales

Se puede encontrar más información sobre distintos paradigmas de programación en la

Wikipedia (http://en.wikipedia.org/wiki/Programming_paradigm) .

3. Abstracción

El concepto de *abstracción* es fundamental en informática. Para modelar un dominio (sistema de información de una universidad, sistema de sensores de una planta química, etc.) es necesario definir distintas *abstracciones* que nos permitan tratar sus elementos.

Una abstracción agrupa un conjunto de elementos (datos y procedimientos) y le da un nombre. Por ejemplo, cuando hablamos del sistema de información de una universidad identificamos elementos como:

- Estudiantes
- Asignaturas
- Matrícula
- Expediente académico
- ...

Existen abstracciones propias de la computación, que se utilizan en múltiples dominios. Por ejemplo, abstracciones de datos como:

- Listas
- Árboles
- Grafos
- Tablas hash

También existen abstracciones que nos permiten tratar con dispositivos y ordenadores externos:

- Fichero
- Raster gráfico
- Protocolo TCP/IP

Uno de los trabajos principales de un informático es la construcción de abstracciones que permitan ahorrar tiempo y esfuerzo a la hora de tratar con la complejidad del mundo real.

Tal y como dice Joel Spolsky en su blog Joel on Software (http://www.joelonsoftware.com/) :

"TCP is what computer scientists like to call an abstraction: a simplification of something much more complicated that is going on under the covers. As it turns out, a lot of computer programming consists of building abstractions. What is a string library? It's a way to pretend that computers can manipulate strings just as easily as they can manipulate numbers. What is a file system? It's a way to pretend that a hard drive isn't really a bunch of

spinning magnetic platters that can store bits at certain locations, but rather a hierarchical system of folders-within-folders containing individual files that in turn consist of one or more strings of bytes."

Una misión fundamental de los lenguajes de programación es proporcionar herramientas que sirvan para construir estas abstracciones.

4. Scheme Como lenguaje de programación

4.1. Algunos ejemplos de Scheme

Scheme es un lenguaje interpretado. Podemos lanzar un intérprete de Scheme y teclear en el *prompt* algunas expresiones. El intérprete analizará la *expresión* y mostrará el *valor* resultante de *evaluarla*.

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

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

La sintaxis es la siguiente:

```
(<función> <arg1> ... <argn>)
```

En Scheme podemos interpretar los paréntesis abiertos '(' como *evaluadores* o *lanzadores* de la función que hay a continuación.

La forma de evaluar una expresión en Scheme es muy sencilla:

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

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

Atención, preguntas

- ¿En qué orden se evalúan los argumentos?
- ¿Influye ese orden en el resultado de la evaluación?

En Scheme los términos *función* y *procedimiento* significan lo mismo y se usan de forma intercambiable. Son ejemplos de funciones o procedimientos: `+`, `-`, `*`.

En Scheme la evaluación de una función siempre devuelve un valor, a no ser que se produzca un error que detiene la evaluación:

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

4.2. ¿Por qué Scheme?

Scheme es un dialecto de Lisp.

Scheme es un lenguaje actual. Existen múltiples proyectos y bibliotecas que se están implementando en la actualidad en Scheme. Algunos ejemplos:

- Colección de programas en Scheme (http://www.rodoval.com/paginalen.php?len=Scheme)
- Scheme Gimp (http://gimp.org.es/tutoriales/schemebasic/) , Scheme está dentro de Gimp para extender la herramienta de tratamiento de imágenes
- Recetas en Scheme (http://schemecookbook.org/Cookbook/WebHome)
- PLaneT (http://planet.plt-scheme.org/) , un repositorio de paquetes escritos en Scheme.

Scheme es un lenguaje ideal para LPP porque:

- Tiene una sintaxis muy sencilla
- Es un lenguaje de script
- Tiene múltiples extensiones: programación orientada a objetos, etc.
- Es posible de extender mediante macros

Existen a su vez múltiples intérpretes de Scheme, nosotros vamos a usar uno de las más extendidos: MzScheme en el entorno de programación DrScheme (http://www.plt-scheme.org/software/drscheme/) (desarrollados por el grupo PLT Scheme (http://www.plt-scheme.org/)). Para una breve introducción al lenguaje y al entorno de programación consultar el manual *A brief tour of DrScheme* (http://www.plt-scheme.org/software/drscheme/tour/) escrito por el mismo grupo.

4.3. Algunas primitivas

Recordemos del apartado anterior que un lenguaje de programación define unos:

- procedimientos primitivos

- mecanismos de composición
- mecanismos de abstracción

Vamos a revisar Scheme desde esta perspectiva.

Las primitivas de Scheme consisten en un conjunto de tipos de datos, formas especiales y funciones incluidas en el lenguaje. A lo largo del curso iremos introduciendo estas primitivas. Es un lenguaje no demasiado complejo (a diferencia de otros como C++ o Java). Está descrito completamente en un documento de especificación de 50 páginas llamado *Revised 5 Report on the Algorithmic Language Scheme* (se puede consultar en este enlace (http://www.schemers.org/Documents/Standards/R5RS/)). Las primitivas del lenguaje están descritas en las 21 páginas del apartado 6 (*Standard procedures*). En la actualidad este documento se encuentra en revisión y en breve se aprobará el estándar 6 (http://www.r6rs.org/) .

Vamos a revisar los tipos de datos primitivos de Scheme, así como algunos procedimientos primitivos para trabajar con valores de esos tipos.

- Booleanos
- Números
- Caracteres
- Cadenas
- Símbolos
- Parejas y listas (*)
- Vectores (*)
- Procedimientos

(*) Los veremos en detalle en futuras clases, cuando hablemos de tipos de datos compuestos. En la clase de hoy sólo veremos unas algunas funciones elementales para crea, recorrer y añadir listas.

Booleanos

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

Atención, aviso

- Nótese en los ejemplos la diferencia entre `(= 3 3.0)` y `(equal? 3 3.0)`.

Números

La cantidad de tipos numéricos que soporta Scheme es grande. Sólo vamos a ver una pequeña parte.

```
number
complex
real
rational
integer
```

Algunos procedimientos primitivos

```
(<= 2 3 3 4 5)
(max 3 5 10 1000)
(/ 22 4)
(quotient 22 4)
(remainder 22 4)
(equal? 0.5 (/ 1 2))
(= 0.5 (/ 1 2))
(abs (* 3 -2))
(floor 3.4)   ; relacionados: ceiling, truncate, round
(sin 2.2)     ; relacionados: cos, tan, asin, acos, atan
```

Caracteres

```
#\a
#\A
#\space
#\ñ         ; se soportan caracteres internacionales
#\á         ; se codifican en UTF-8
(char<? #\a #\b)
(char-numeric? \#1) ; relacionados: char-alphabetic?
                    ; char-whitespace?, char-upper-case?
                    ; char-lower-case?
(char-upcase #\ñ)
(char->integer #\space)
```

Cadenas

Las cadenas son secuencias finitas de caracteres.

```
"hola"
"La palabra \"hola\" tiene 4 letras"
(make-string 10 #\o)
(substring "Hola que tal" 2 4)
(string? "hola")
(string->list "hola")
```

Símbolos

Un símbolo es lo que en otros lenguajes se denomina *identificador*. En Scheme los símbolos

e identificadores pueden contener caracteres internacionales (Unicode, UTF-8). El intérprete DrScheme guarda los ficheros de texto en esa codificación.

```
'hola
(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
(equal? 'hola 'hola)
(equal? 'hola "hola")
```

Diferencia entre símbolos y cadenas: un símbolo (identificador) es un objeto simple y una cadena es un objeto compuesto. El intérprete de Scheme codifica un símbolo mediante un único número, su *valor hash*. Otra diferencia bastante clara: un símbolo no puede contener un espacio, pero una cadena sí.

Listas

Uno de los elementos fundamentales de Scheme, y de Lisp, son las listas. Vamos a ver cómo definir, crear, recorrer y concatenar listas.

```
(list 1 2 3 4)     ; list crea una lista
'(1 2 3 4)         ; otra forma de definir la misma lista
(car '(1 2 3 4))   ; primer elemento de la lista
(cdr '(1 2 3 4))   ; resto de la lista
'()                ; lista vacía
(cdr '(1))         ; devuelve la lista vacía
(null? (cdr '(1))) ; comprueba si una lista es vacía
(append '(1) '(2 3 4) '(5 6)) ; construye una lista nueva concatenando
                              ; los argumentos
```

Atención, aviso

Uno de los conceptos más importantes relacionados con las listas es el de lista vacía. La comprobación de si una lista es vacía con la función `null?` es el caso base de gran parte de funciones recursivas que recorren listas.

Procedimientos

Una característica fundamental de Scheme es que los procedimientos son tipos primitivos. Más adelante veremos esto en detalle.

```
+
-
remainder
```

4.4. Extensiones de Scheme

Es posible definir nuevos procedimientos en Scheme (lo explicamos en el siguiente apartado). Vamos a usar también un conjunto de procedimientos definidos en el fichero simply.scm (programas/simply.scm.zip) . Son procedimientos definidos en el libro '*Simply Scheme: Introducing Computer Science*' de Brian Harvey and Matthew Wright.

Los procedimientos que vamos a usar más a menudo son procedimientos para tratar con símbolos y cadenas (composición, primer carácter, etc.): `first`, `butfirst` (`bf`), `last`, `butlast` (`bl`), `word`, `empty?` y `member?`. Además define la función `1+` que incrementa en 1 su argumento.

```
(load "/home/domingo/simply.scm") ; en Windows: (load "C:\simply.scm")
(first 'hola)
(butfirst 'hola)
(bf 'hola)
(last 'hola)
(butlast 'hola)
(word 'hola 'mundo)
(word 12 34)
(+ (first 23) 5)
(empty? "hola")
(empty? (bf "h"))
(empty? (bf 'h))
(member? 'f 'abcdefg)
true
false
(1+ 4)
```

4.5. Abstracción: define

Las capacidades de *abstracción* de un lenguaje de programación hacen posible definir nuevos elementos y darles un nombre. En otros lenguajes de programación (Java, C++, etc.) es posible definir nuevos tipos de datos (clases), definir procedimientos y funciones de esos tipos de datos, definir constantes, etc.

En Scheme sólo tenemos una instrucción para definir nuevos elementos: `define` (más adelante veremos que esto no es completamente cierto, ya que podemos definir *macros* con la instrucción `define-syntax`).

Podemos usar `define` para darle un nombre (identificador, símbolo) a un valor. Una vez definido, podemos usar el nombre en lugar del valor. Ya veremos más adelante que esto es equivalente a decir que el símbolo *se evalúa* al valor.

```
> (define pi 3.14159)
```

```
> pi
3.14159
> (sin (/ pi 2))
0.9999999999991198
> (define a (+ 2 (* 3 4)))
> a
14
```

La sintaxis de `define` es:

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

La forma especial `(define` *símbolo expresión*`)` se evalúa así:

1. Evaluar *expresión*
2. Asociar el valor resultante de la evaluación anterior con el *símbolo*

El otro uso de `define` es para definir nuevos procedimientos (aunque ya veremos más adelante que Scheme utiliza siempre el define anterior, y que esta forma del define no es más que *azucar sintáctico*).

La sintaxis para definir un procedimiento es:

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

Se define una función con el nombre, argumentos y cuerpo dados. El cuerpo es una expresión de Scheme. El resultado de evaluar la última expresión del cuerpo es el valor devuelto por al función.

Ejemplos:

```
> (define (cuadrado x)
   (* x x))
> (cuadrado 2)
4
> (cuadrado 3 4 5) ; error

(define (plural pal)
   (word pal 's))
> (plural 'perro)
perros
> (plural 'arbol)
arbols
```

Algunos ejemplos curiosos (en Scheme los símbolos no están protegidos):

```
(define suma +)
(suma 1)
(suma 1 2 3)
(define + -)
(+ 2 3)
```

```
(suma 2 3)
(define + suma)
```

Estos ejemplos explican un poco más la idea de que en Scheme los procedimientos son valores primitivos. Al hacer `(define + -)` estamos asociando el identificador '+' al procedimiento 'resta' que resulta de evaluar el identificador '-'.

4.6. Estructuras de control

El otro elemento comúm a todos los lenguajes de programación (aparte de la abstracción y de las primitivas) es la posibilidad de *componer* expresiones sencillas en expresiones compuestas.

Hemos visto que una de las características de Scheme es la composición de funciones. Scheme también define *estructuras de control* que nos permiten seleccionar qué parte de una expresión evaluamos en función del resultado de la evaluación de otras.

Las estructuras más importantes de Scheme son el `if` y el `cond` para realizar una evaluación condicional.

El nombre que reciben en Scheme todas las expresiones que no son procedimientos y que tienen una evaluación especial es *forma especial*. Ejemplos de formas especiales son: `define`, `if`, `cond`.

Atención, aviso

- La expresión `(+ (* 2 3) 4)` se evalúa de una forma distinta de `(define a 3)`. En el primer caso estamos evaluando llamadas a funciones, y en el segundo evaluamos una forma especial.

4.6.1. Forma especial 'if'

La forma especial `if` realiza una evaluación condicional de las expresiones que la siguen, según el resultado de una condición.

```
(define x 3)

(if (> x 5)
   'mayor-que-cinco
   'menor-o-igual-que-cinco)

(define (vocal? x)
   (member? x 'aeiou))

(define (plural pal)
```

```
(if (vocal? (last pal))
   (word pal 's)
   (word pal 'es)))
```

Para que un programa Scheme sea legible es muy importante la indentación (tabulación) correcta de sus expresiones.

La sintaxis de `if` es:

```
(if <condición> <exp-verdad> <exp-falso>)
```

El funcionamiento de la forma especial es:

1. Evaluar *condición*
2. Si el resultado es true evaluar *exp-verdad*
3. Sino evaluar *exp-falso*

```
(if (> 3 2) (* 2 3) (/ 2 0))
```

En la expresión anterior la condición es verdadera, por lo que no se evalúa la expresión `(/ 2 0)` que daría un error. Se evalúa sólo `(* 2 3)` y se devuelve su resultado.

4.6.2. Forma especial 'cond'

La forma especial `cond` evalúa una serie de condiciones y devuelve el valor de la expresión asociada a la primera condición verdadera.

```
(cond
   ((> 3 4) '3-es-mayor-que-4)
   ((< 2 1) '2-es-menor-que-1)
   ((= 3 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))

; juego del ding:
1,2,ding,4,5,ding,7,8,ding,10,11,ding,ding,14,ding,...

(define (ding x)
   (cond
      ((= (remainder x 3) 0) 'ding)
      ((member? 3 x) 'ding)
      (else x)))
```

La sintaxis de `cond` es la siguiente:

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

La semántica es la siguiente:

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'

Se hace notar

- En cualquier caso, sólo se evalúa un único consecuente del `cond`.

4.6.3. ¿Y los bucles?

En programación funcional pura no existe el concepto de bucle tan extendido en los lenguajes de programación imperativos. Ya veremos que no es necesario si tenemos la recursión. Cualquier función en donde se necesite repetir una sentencia un número de veces (o hasta que se cumpla una determinada condición) se puede expresar de forma recursiva.

Aunque en Scheme existe la forma especial `(do ...)` que permite implementar un bucle nosotros no vamos a usarla. El uso de esta forma especial nos aparta del paradigma funcional, ya que su semántica no puede definirse con el modelo de sustitución que veremos en las próximas clases.

Por ejemplo, las siguientes funciones son equivalentes. En C, usando un bucle `while`

```
suma-hasta (k) {
   suma=0;j=0;
   while (j<=k) {
      suma = suma+j;
      j++;
   }
}
```

En Scheme, usando la recursión:

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

5. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Encyclopedia of Computer Science (Wiley, 2000). Disponible en la biblioteca politécnica (POE R0/E/I/ENC/RAL). Consultar las entradas:
  - Control structures
  - Lisp
  - Programming
  - Program
  - Programming languages
- Árbol genealógico de los lenguajes de programación (http://www.levenez.com./lang)
- Abstraction (http://en.wikipedia.org/wiki/Abstraction_%28computer_science%29) (Wikipedia)
- Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , Abelson y Sussman, MIT Press 1996 (pp.1-13). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))
- DrScheme (http://www.plt-scheme.org/software/drscheme/)
- A brief tour of DrScheme (http://www.plt-scheme.org/software/drscheme/tour/)