Tema 2: Introducción a Scheme

Sesión 3: Introducción a Scheme (1)

Referencias

- DrRacket (http://racket-lang.org/)

- A brief tour of DrScheme (http://www.plt-scheme.org/software/drscheme/tour/)

- 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), en concreto los capítulos 1.1.1.-1.1.4, 1.1.6 SICP: The elements of programming: Expressions, Evaluation Combinations, Conditional Expressions and Predicates, Naming and Environment, Compound procedures

- Teach yourself Scheme (versión HTML, versión PDF)

- Simply Scheme online

Scheme como lenguaje de programación

- Dialecto de LISP
- MIT 1975, Guy L. Steel y Gerarld J. Sussman
- Lenguaje académico, ligado a cursos de introducción a la computación en universidades de EEUU (Libro SICP, Abelson y Sussman)
- Lenguaje para iniciar a la programación en los institutos en EEUU (Bootstrapworld o How to Design Programs)

Scheme es actual

- 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.

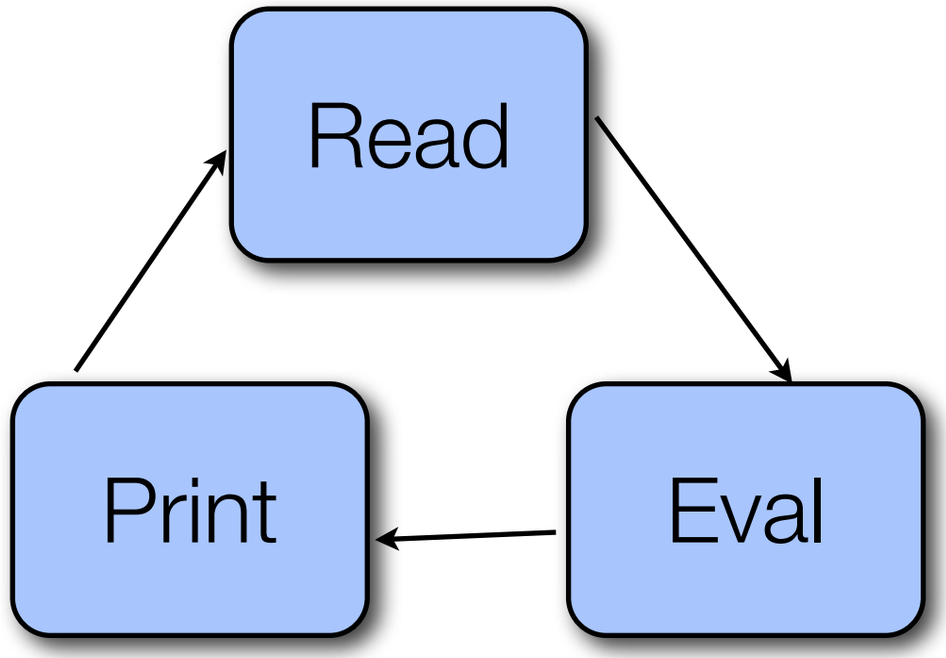
Elementos de un lenguaje

- Primitivas
- Mecanismos de composición
- Mecanismos de abstracción

Lenguaje interpretado

- Arrancamos DrRacket
- Vamos a probar algunas expresiones

```
(+ 2 3)
(+ (* 2 3) (+ 1 2 3 4) (/ 12 3))
(> 3 (+ 2 5))
(and (> 3 1) (= (* 2 3) (/ 12 2)))
(string-append "hola" "adios")
```



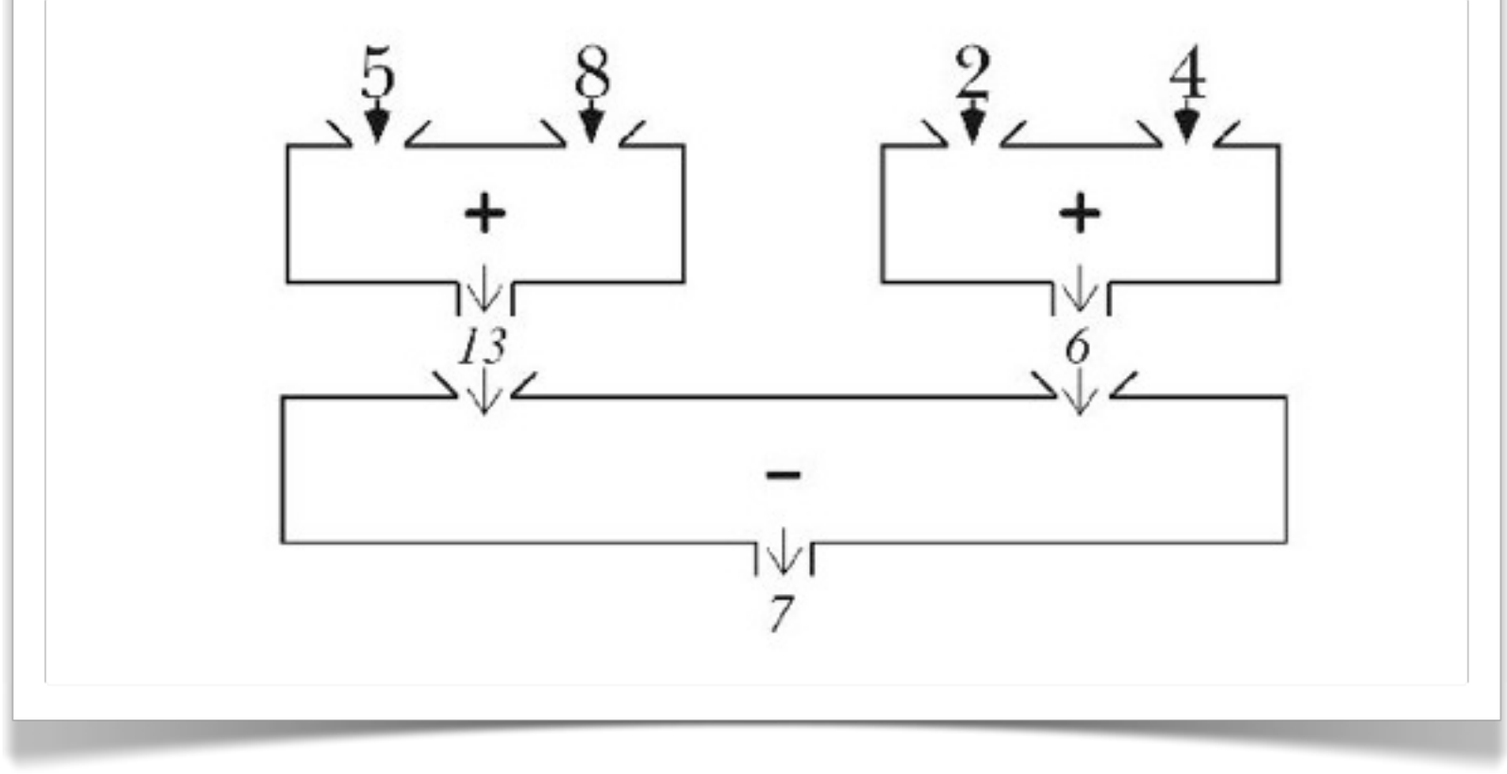
Composición de expresiones

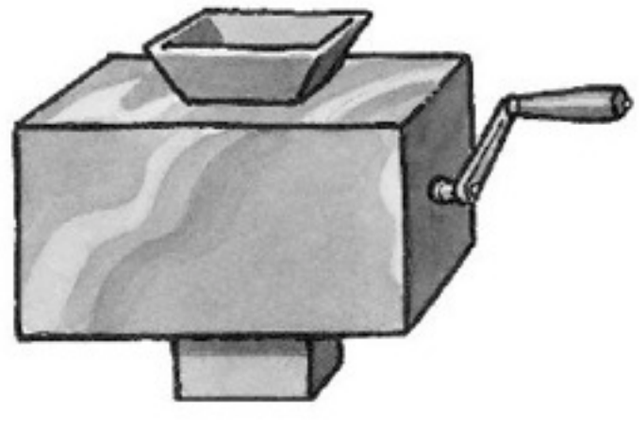
- Primitivas = +, *, >, =, and, string-append, …
- Valores = números, booleanos, strings, …
- Un paréntesis abierto '(' lanza la función que hay a su derecha
- Composición = anidar expresiones

Evaluación de expresiones

- Paréntesis: evaluación de procedimientos

```
(- (+ 5 8) (+ 2 4))
```





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)
```

Números

```
number
complex
real
rational
integer
```

```
(<= 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))
(sin 2.2)
```

```
(number? 1)
(integer? 2.3)
(integer? 4.0)
(real? 1)
(positive? -4)
(negative? -4)
(zero? 0.2)
(infinite? 2.0)
(finite? +inf.0)
(even? 2)
(odd? 3)
(exact? 5)
(inexact? +inf.0)
```

Caracteres

- Se soportan caracteres internacionales y se codifican en UTF-8

```
#\a
#\A
#\space
#\ñ
#\á
```

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

Cadenas

```
(make-string 5 #\o) --> "ooooo"
(string #\h #\o #\l #\a) --> "hola"
(substring "Hola que tal" 2 4)
(string? "hola")
(string->list "hola")
(string-length "hola")
(string-ref "hola" 0)
(string-append "hola" "adios")
(string=? "Hola" "hola")
(string=? "hola" "hola")
(string<? "aab" "cde")
(string>=? "www" "qqq")
```

Símbolos

```
'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
(symbol?  #f) ; #f
(equal? 'hola 'hola)
(equal? 'hola "hola")
```

Abstracción: Forma especial **define** para dar valores a variables

Sintaxis:

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

Semántica:

1. Se evalúa la <expresión>
2. El resultado de la evaluación queda asociado al <símbolo>

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

Abstracción: Forma especial **define** para crear funciones

Sintaxis:

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

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

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

Ejercicio

- Define una función que devuelva la mitad de una cadena. Ejemplo:

```
(mitad-cadena “hola”)
“ho”
(mitad-cadena “pepito”)
“pep”
```

Solución

```
(define (mitad-cadena cad)
    (substring cad 0
               (quotient (string-length cad) 2)))
```

Estructuras de control: Forma especial **if**

Sintaxis:

```
(if condicion expresion-true expresion-else)
```

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

Sólo se evalúa la expresión asociada al resultado de la condición
Los siguientes ejemplos no darían error:

```
(if (> 7 5) (+ 2 1) (/ 3 0))
(if (< 7 5) (gfdgfg 2) (+ 2 2))
```

Estructuras de control: Forma especial **cond**

Sintaxis:

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

```
(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))
```