Fundamentos de lenguajes de programación

Robinson Duque, Ph.D

Universidad del Valle

robinson.duque@correounivalle.edu.co

Programa de Ingeniería de Sistemas
Escuela de Ingeniería de Sistemas y Computación

Este documento es una adaptación del material original de los profesores
Carlos Andres Delgado y Carlos Alberto Ramírez

Overview

Notas sobre Dr Racket

- ¡Lo puedes instalar en Windows, Linux y Mac OS !
- Se recomienda trabajar con la versiones 7.2 o 7.3, algunas versiones como la 6.2 tienen problema con la librería SLLGEN
- Las expresiones en Dr Racket son en notación prefija, por ejemplo (+ 5 2) es equivalente 5+2 en notación infija.
- Se recomienda utilizar paréntesis para evitar problemas en la interpretación de resultados por ejemplo (+ (- 3 2) 4) es equivalente en notación infija a ((3 - 2) + 4)
- Para el curso de Fundamentos de Lenguajes de programación debe seleccionar **The Racket Lenguaje** y agregar en la primera linea *#lang eopl* .

Notas sobre Dr Racket

- Se utiliza la palabra reservada **define** para la definición de variables por ejemplo

```
(define numeroA 5)
(define numeroB (* 2 numeroA))
```

- Se definen las funciones de la siguiente forma:

```
(define (nombreFuncion argumentosEntrada) <operaciones>)
```

Ejemplo:

```
(define (multiplique a b) (* a (* b b)))
```

¿Que debería retornar (multiplique 3 9)?.

Notas sobre Dr Racket

Así mismo, se pueden definir funciones anónimas con la expresión **lambda**. Ejemplo:

```
(lambda (x y) (if (> x y) (* x y) (- x y)))
```

Este retorna un valor función o procedimiento, estos podemos evaluarlos así:

```
( (lambda (x y) (if (> x y) (* x y) (- x y))) 3 4)
```

Observe que la función espera 2 argumentos.

Notas sobre Dr Racket

Importante

Debido a que en el paradigma funcional no hay diferencia entre funciones y valores, usaremos lambda para representar las funciones como valores.

Para el ejemplo anterior:

```
(define multiplique
  (lambda (a b)
          (* a (* b b))
    )
)
```

Notas sobre Dr Racket

Ejercicio

Escriba en el Dr Racket utilizando notación prefija:

- $2 * 2 + 3 * 5 + (\frac{1}{4})^2$
- $2 * (1 + 3^2 + \frac{4}{4}) + 3 * (5 - 3) + \frac{12}{4} - 3 + 4 * 5^3$

Diseñe una función anónima que reciba dos valores. Si ambos valores son positivo o ambos son negativos retorna su multiplicación, en caso contrario retorna su suma.

Notas sobre Dr Racket

Ejercicio 1

$2 * 2 + 3 * 5 + (\frac{1}{4})^2$

```
(+ (* 2 2) (* 3 5) (expt (/ 1 4) 2))
```

Respuesta: 19.0625

Notas sobre Dr Racket

Ejercicio 2

$$2 * (1 + 3^2 + \frac{4}{4}) + 3 * (5 - 3) + \frac{12}{4} - 3 + 4 * 5^3$$

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

Respuesta: 528

Notas sobre Dr Racket

Ejercicio 3

```
(lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y)))
```

Probemos

```
( (lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y))) 2 3)
( (lambda (x y) (if (>= (* x y) 0) (* x y) (+ x y))) 2 -3)
```

Condicionales

En Racket tenemos el condicional que usualmente trabajamos en diferentes lenguajes

```
(if pregunta hacer_si_verdad hacer_si_falso)
```

Ejemplo:

```
(define funcionMisteriosa
  (lambda (n l)
    (if (< n l) "n es menor que l" "n es mayor o igual que l")
  )
```

Condicionales

También, contamos con condicionales que permiten verificar varias condiciones sucesivamente. Sólo se da respuesta a la primera que sea verdadera, si ninguna lo es se responderá lo indicando el else.

```
(cond
        [Pregunta Respuesta]
        [Pregunta Respuesta]
        ...
        [else Respuesta]
)
```

Condicionales

Por ejemplo, una función que recibe un número y verifica si es par

```
(define verificarpar
  (lambda (n)
        (cond
                [(even? n) "Par"]
                [(< n 0) "Impar y menor que 0"]
                [else "Impar y mayor o igual que 0"]
        )
  )
)
```

Condicionales

Diseñe una función que recibe un número y un símbolo

1. Si el número es menor o igual que 0 retorna el símbolo 'error
2. Si no, Si el número es mayor o igual 0 retorna:
   - Si el símbolo es 'm, retorna 'hombre
   - Si el símbolo es 'f, retorna 'mujer
   - En otro caso retorna 'error

Definiciones locales

En algunos casos requerimientos realizar definiciones dentro de funciones, en Fundamentos de programación contábamos con la sentencia **local**, pero para este curso usaremos dos

- **let** para definiciones no recursivas. Dentro de este las funciones no se conocen a si mismas

```
(let (
        ;Zona de declaraciones
        ( .. ) ;Declaración 1
        ( .. ) ;Declaración 2
      )
      ;Zona de expresiones
)
```

- **letrec** para definiciones recursivas. En este las funciones se conocen a sí mismas. Su sintaxis es igual a la de let.

Definiciones locales

```
(define funcion
        (lambda (n)
                (let
                   (
                        (p 2)
                        (f (lambda (a b) (* a b)))
                   )
                   (f n p)
                 )
        )
)
(funcion 10)
```

Definiciones locales

```
(define funcion
        (lambda (n)
                (letrec
                   (
                        (p 2)
                        (sume
                            (lambda (a b)
                             (if (= b 0) 0 (+ a (sume a ( -b 1))))
                            )
                         )
                   )
                   (sume n p)
                 )
        )
)
(funcion 10)
```

Si utilizas let en lugar de letrec, va indicar que no conoce la función suma.

Definiciones locales

Aveces requerimos que un valor declarado conozca los anteriores y que las funciones no se conozcan a si mismas, para esto usaremos let*.

```
(define funcion
        (lambda (n)
                (let*
                   (
                        (p 2)
                        (q 3)
                        (r (+ p q n))
                   )
                   r
                 )
        )
)
(funcion 10)
```

Este código es funcional cambiando let* por letrec, la diferencia radica en las funciones recursivas.

Listas

Las listas son una estructura recursiva, la cual consta de dos partes:

1. Un valor (cualquiera representado por scheme)
2. Una lista: La cual puede ser vacía.

```
(cons 1 (cons 2 (cons 3 empty)))
```

Dr Racket, provee la función list, la cual permite construir listas sin pensar en la recursividad. A esto se le conoce como azúcar sintáctico.

```
(list 1 2 3)
```

Listas

Para acceder a los elementos se tiene. car accede al primer elemento y cdr al resto, el cual es una lista.

```
(define l (cons 1 (cons 2 (cons 3 empty))))
(car l)
(cdr l)
```

Si observa, cdr devuelve una lista. Para acceder al segundo elemento tenemos cadr (primero del resto). Así mismo se tienen funciones similares.

Listas

A lo largo del curso vamos a utilizar una representación de listas especial:

```
'()
```

Esto es una lista vacía, podemos hacer lo siguiente:

```
'(perro gato 1 2 3)
```

Observe que perro y gato no son variables si no que se convierten en símbolos, también puede representar listas dentro de listas así:

```
'( (perro 1 2 3) (3 2 4) gato)
```

Observe que las listas sólo se declaran entre paréntesis. Esta representación permite agilizar el diseño de ciertas funciones.

Ejercicio

Enunciado

Diseñe una función que almacene los factoriales (en una lista) desde 0! hasta un valor n ingresado por el usuario.
Para esto necesitas usar locales (recursivos) ya que se requiere una función que genere una lista entre 0! y n!, la función a diseñar sólo recibe como argumento n.

Ejercicio

```
;Calcula el factorial de un numero
(define factorial
 (lambda (n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))]
    )
  )
)
;Genera una lista de factoriales desde 0! hasta n!
(define lista-factorial
  (lambda (n)
        (letrec
                (
                   (lista-factorial-aux
                     (lambda (n acc)
                        (if (= acc n)
                          (factorial n)
                                  (cons (factorial acc) (lista-factorial-aux n
                                       (+ 1 acc)))
                                )
                          )
                     )
                )
                (lista-factorial-aux n 0)
        )
  )
)
(lista-factorial 10)
```

Funciones como ciudadanos de primera clase

Ejemplo

¡Las funciones pueden ingresar cómo parámetros!

```
(define predicado
  (lambda (a b)
        (cond
                [(< a b) #T]
                [(even? a) #T]
                [else #F]
        )
  )
)

(define filtro
  (lambda (lst num f)
        (cond
                [(eqv? lst '()) empty]
                [(f (car lst) num) (cons (car lst) (filtro (cdr lst) num f))]
                [else (filtro (cdr lst) num f)]
        )
  )
)

(filtro (list 1 2 3 4 5) 3 predicado)
```

¿Que hace esta función?

Funciones como ciudadanos de primera clase

Ejemplo

¡Pueden retornarse funciones!

```
(define funcionMisterio
    (lambda (a b)
        (lambda (x y) (+ x y a b))
    )
)
;Probar
(funcionMisterio 1 2)
( (funcionMisterio 1 2) 3 4)
```

Esto significa que no hay diferencia entre los valores y funciones.

Son exactamente lo mismo

Funciones como ciudadanos de primera clase

1. Diseñe una función, que reciba un número a y una función f. La función f recibe un número y retorna un booleano. Se hace el llamado **(f a)** y si el resultado es verdadero se retorna "ok", en otro caso "falso".
2. Diseñe una función que reciba dos números a y b y retorna una función t la cual espera un argumento numérico s. t evalúa si a es mayor que b si es así retorna $2 * s$ en otro caso $-2 * s$

Asignación y secuenciación

Hasta ahora hemos trabajando en un paradigma declarativo funcional en el cual las variables están ligadas a un valor. Ahora, para trabajar en un paradigma imperativo debemos incluir la noción de estado con la instrucción set!, lo que permite a lo largo del código las variables puedan cambiar un valor. Este **estado** caracteriza la ejecución de un programa. Ejemplo:

```
(define valor 1)
(set! valor 2)
```

En este punto el estado de valor ha cambiado de 1 a 2 durante la ejecución.

Asignación y secuenciación

Otra característica de un paradigma imperativo es la secuenciación, es decir la ejecución de más de una instrucción en un paso dado, ya que el declarativo sólo permite uno. Para esto contamos con begin, este tiene la siguiente sintaxis:

```
(begin
    ;expresion
    ;expresion
    ...
    ;expresion
)
```

Este retorna la última expresión ejecutada, la ventaja es que permite ejecutar más de una expresión en un paso.

Asignación y secuenciación

Otra característica de un paradigma imperativo es la secuenciación, es decir la ejecución de más de una instrucción en un paso dado, ya que el declarativo sólo permite uno. Para esto contamos con begin, este tiene la siguiente sintaxis:

```
(define funcion
    (lambda (a b)
      (begin
        (set! a (* a b))
        (set! b (- a b))
        (+ a b)
       )
     )
)
(funcion 2 3)
```

Observe que se retorna la ultima expresión ejecutada.

Próxima sesión

- Relación entre inducción y programación (Capitulo 1 EOPL).