Introducción a la Programación Funcional

lunes, 29 de noviembre de 2021 21:38

- Programación Funcional
  - Es un paradigma diferente a los imperativos
  - Está basado en ***funciones matemáticas***
  - ***LISP*** es el primer lenguaje funcional, del cual derivas ***Scheme, Common, LISP, ML*** y ***Hasket***

- Funciones Matemáticas
  - Una función es una ***proyección de un conjunto dominio a otro que es el rango***
  - $f: D \to R$
  - La evaluación de funciones está controlada por ***recursión*** y ***condiciones***
  - Las funciones matemáticas siempre entregan ***el mismo valor*** para el mismo conjunto de argumentos, o sea, ***no tiene efectos laterales***
  - Ejemplos
    - Definición de una función matemática: $cubo(x) = x \cdot x \cdot x$ con *x* real
    - Aplicación de la función matemática: $cubo(2.0) \to 8.0$
    - Definición de una función lambda: $\lambda(x)x \cdot x \cdot x$
    - Aplicación de una función lambda: $(\lambda(x)x \cdot x \cdot x)(2.0) \to 8.0$

- Formas Funcionales (Funciones de Primer Orden)
  - Toman ***funciones como parámetros y/o producen funciones como resultado***
  - Composición de funciones
    - $h = f \circ g$ entonces $h(x) = f(g(x))$
  - Construcción
    - Lista de funciones que se aplican a un mismo argumento
    - $[f, g](x)$ produce $(f(x), g(x))$
  - Aplicación a todo
    - Una misma función se aplica a una lista de argumentos
    - $\alpha\left(f, (x, y, z)\right)$ produce $\left(f(x), f(y), f(z)\right)$

(operador arg1 arg2 ...)

- Fundamentos de la Programación Funcional
  - La programación funcional ***no usa variables*** ni asignación
  - La repetición debe ser lograda con la ***recursión***
  - Un ***programa*** consiste en la definición de funciones y las aplicaciones de estas
  - La ***ejecución*** es la evaluación de las funciones
  - La ***transparencia referencial*** se refiere a que la evaluación de una función siempre producirá ***el mismo resultado***

- Lenguajes Funcionales
  - El lenguaje provee algunas funciones básicas, que son primitivas para la construcción de funciones más complejas
  - Se definen algunas estructuras para representar datos de los parámetros y resultados de las funciones

Scheme

lunes, 29 de noviembre de 2021 23:36

- Características
  - Pequeño, con sintaxis y semántica simple
  - Nombres tiene solo ámbito estático (Solo existe dentro de los paréntesis donde se define)
  - Las funciones son ***entidades de primera clase***, por ende se tratan como cualquier valor
  - Recolección automática de basura

- Ambiente Interactivo
  - Corresponde al ciclo "Leer -> Evaluar -> Imprimir" denominado ***REPL***
  - El sistema entrega un pronto, se ingresa la expresión, el sistema evalúa y entrega el resultado

- Identificadores
  - Corresponde a ***palabras claves, variables*** y ***símbolos***, que no son sensibles a las mayúsculas
    - Se forman de:
      - mayúsculas y minúsculas ['A' .. 'Z', 'a'..'z']
      - dígitos ['0'..'9']
      - caracteres [? ! . + - * / < = > : $ % ^ & _ ~]
  -
    - Identificadores no pueden comenzar un número.
      - Válidos: X3, ?$!!!, Abcd, AbcD
      - No lo es: 8id

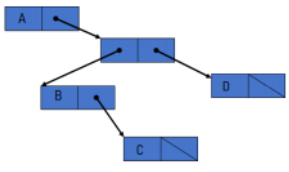
- Constantes Básicas
  - String
    - Se escribe usando comillas dobles
  - Carácter
    - Lo precede un #
  - Número
    - Pueden ser enteros, fracciones, punto flotante y notación científica
  - Números Complejos
    - Coordenadas rectangulares y polares
  - Booleanos
    - Valores falso (#f) o verdadero (#t)

- Funciones Aritméticas
  - Los nombres ***+,-,**** y ***/*** son nombres reservados para op. Aritméticas
    - Funciones se escriben como listas en notación prefija:
  -
    (+ 1/2 1/2) => 1
    (- 2 (* 4 1/3)) => 2/3
    (/ (* 6/7 7/2) (- 4.5 1.5)) => 1.0

- Listas
  - Se escriben con paréntesis redondos
    - (a b c d)
  - Contienen elementos de cualquier tipo los cuales se pueden anidar
    - (lambda(x)(* x x))
  - Una función se escribe como una lista en notación prefija, o sea, el primer elemento de la lista es la función y el resto son los argumentos
    - (+ 3 2) -> 5
  - Toda lista es evaluada, a menos que, con un ***quote***, se indique lo contrario
    - '(1 2 3 4) -> (1 2 3 4)
    - (quote (1 2 3 4)) -> (1 2 3 4)
  - Operadores Básicos de Listas
    - Car
      - Devuelve el primer elemento de una lista
      - (car '(a b c d)) -> a
      - ***First*** es análogo a car
    - Cdr
      - Devuelve el resto de una lista, o sea, todo menos el primer elemento
      - (cdr '(a b c d)) -> (b c d)
      - ***Rest*** es análogo a cdr

Estructura de una Lista

(A (B C) D)



  - Constructores
    - Cons
      - Construye una nueva lista cuyo ***car*** y ***cdr*** son los dos argumentos

    - (cons ´a ´(b c d)) => (a b c d)
    - (cons (car ´(a b c))(cdr ´(a b c)) => (a b c)
  - List
    - Construye una lista con todos los argumentos
    - (list ´a ´b ´c ´d) => (a b c d)
    - (list) => ()
  - Append
    - Fusiona dos listas en una sola, pero no altera las listas que se dan como argumento
    - (append ´(a b) ´(c d)) => (a b c d)

- Let
  - Permite definir variables que se ligan a un valor en una evaluación de expresiones
    • Sintaxis:

      (let ((var1 val1) …) exp1 exp2 …)

  - • Ejemplo:

      (let ((x 2) (y 3))
        (* (+ x y) (- x y))) => -5

      Importante: las variables sólo tienen ámbito local

- Expresiones Lambda
  - Permite crear un nuevo procedimiento
    • Sintaxis:

      (lambda (var1 var2 …) exp1 exp2 …)

  - • Ejemplo:

      ((lambda (x) (* x x)) 3) => 9

      ¡Una expresión lambda es un objeto tipo procedimiento que no tiene nombre!

  - Ejemplo:

```
(let ((square (lambda (x) (* x x))))
                    (list (square 2)
                          (square 3)
                          (square 4)
                    )
)

=> (4 9 16)
```

Scheme: relación entre let y lambda

• Nótese que:

(let ((var$_1$ val$_1$) … (var$_m$ val$_m$)) exp$_1$ … exp$_n$)

equivale a:

((lambda (var$_1$ … var$_m$) exp1 … exp$_n$) val$_1$ … val$_m$)

- Especificación de parámetros formales
  - • Lista propia de parámetros (var$_1$ var$_2$ … var$_n$)

    ((lambda (x y) (list x y)) 1 2) => (1 2)

  - • Parámetros único var$_r$

    ((lambda x (list x)) 1 2) => ((1 2))

  - • Lista impropia de parámetros (var$_1$ var$_2$ … var$_n$. var$_r$)

    ((lambda (x . y) (list x y)) 1 2 3) => (1 (2 3))

- Definiciones de nivel superior
  - Las variables definidas con ***let*** y ***lambda*** son visibles solo en el cuerpo de las expresiones (local)
  - El procedimiento ***define*** permite definir variables de nivel superior (global)
  - Estas definiciones permiten visibilidad en cada expresión donde no sean escondidas por otro ligado
    - Ej: Una variable definida con el mismo nombre mediante ***let*** oculta a las de nivel superior
  - • Uso de **define**:

    (define pi 3.1416) => pi

  - (define square (lambda (x) (* x x)))) => square

    (square 3) => 9

    (let ((x 2)(square 4)) (* x square)) => 8

- La forma: (**define** var_0 (**lambda** (var_1 ... var_n) e_1 ...))
  se puede abreviar como: (**define** (var_0 var_1 ... var_n) e_1 ...)

- **Ejemplo:** las siguientes expresiones son equivalentes
  (**define** square (**lambda** (x) (* x x))))
  (**define** (square x) (* x x))

Condicionales

martes, 30 de noviembre de 2021 0:36

- En Scheme es posible condicionar la realización de una determinada tarea
  - Sintaxis: (if test consecuencia alternativa)
  - Ejemplo:
    - ```
      (define (abs n) (if (> n 0)
                          n
                          (- 0 n)))

      (abs -27)      => 27
      ```

- Condicionales Múltiples
  - Expresiones que evalúan condicionalmente
    - Sintaxis:
    - ```
      (cond
          (test1 exp1)
          (test2 exp2) ...
          (else exp_n)
      )
      ```
    - El uso de **else** es opcional, siendo equivalente su uso a colocar **#t**.
    - Ejemplo:
    - ```
      (define abs2
        (lambda (x)
                (cond  ((= x 0) 0)
                       ((< x 0) (- 0 x))
                       (else x)
                )
        )
      )
      ```

- Predicados
  - Procedimientos para expresiones relacionales: =, <, >, <= y >=
    Ejemplo: (= 3 4) => #f
  - Procedimientos para expresiones lógicos: **or**, **and** y **not**
    Ejemplo: (**and** (> 5 2) (< 5 10)) => #t
  - Lista nula: **null?** (null? '()) => #t
  - Argumentos equivalentes: **eqv?** (eqv? 'a 'a) => #t
  - Ejemplo:
    - ```
      (define (reciproco n)
              (if  (and (number? n)  (not (= n 0)))
                   (/ 1 n)
      ```

- Lista nula: **null?** (**null?** ´()) => #t

- Argumentos equivalentes: **eqv?** (**eqv?** ´a ´a) => #t

- **Ejemplo:**

```
(define (reciproco n)
        (if  (and (number? n)  (not (= n 0)))
             (/ 1 n)
             "reciproco: división no válida")
        )
)
```

- Cualquier objeto se interpreta como #t

- La lista nula '() equivale a #f (sólo en Estándar IEEE)

- Se definen los predicados:
  - **pair?** : verifica si es un par (lista propia o impropia)
  - **number?** : verifica si es número
  - **string?** : verifica si es un string

Recursión

martes, 30 de noviembre de 2021 14:33

- Tipos de Recursión
  - Directa
    - La función se invoca a sí misma
    - ```
      (define length
        (lambda (ls)
          (if (null? ls)
              0
              (+ 1 (length (cdr ls))))))
      => length

      (length '(a b c d))                    => 4
      ```
    - ```
      ;; El siguiente procedimiento busca x en la lista ls,
      ;; devuelve el resto de la lista después de x o ()

      (define memv
        (lambda (x ls)
          (cond ((null? ls) ())
                ((eqv? x (car ls)) (cdr ls))
                (else (memv x (cdr ls))))))
      => memv

      (memv 'c '(a b c d e))          => (d e)
      ```
  - Indirecta
    - La función invoca a otra función, y quizá está a otras, las que terminan invocando a la primera
  - Lineal
    - Existe una única invocación recursiva
  - Múltiple
    - Existe más de una invocación recursiva
    - Anidada
      - Dentro de una invocación recursiva se tiene como parámetro otra invocación recursiva
  - De Cabeza
    - La invocación recursiva es lo primero que se hace
    - Ej: Post orden
  - Intermedia
    - Las sentencias aparecen antes y después de la invocación recursiva
  - De Cola
    - La invocación recursiva se hace después de todas las sentencias

- Recursividad de Cola
  - Tipo de Recursión Directa
  - Cuando un ***llamado*** a procedimiento aparece al final de una expresión lambda, es un ***llamado de cola***, o sea no debe quedar nada por evaluar
  - ***Recursión de cola*** es cuando un procedimiento hace un llamado de cola ***hacia si*** mismo
  - Son llamados de cola a f:
    ```
    (lambda () (if (g) (f) #f))
    (lambda () (or (g) (f)))
    ```
    pero no lo son respecto a g:
    ```
    (lambda () (if (g) (f) #f))
    (lambda () (or (g) (f)))
    ```
  - Propiedades
    - ***Scheme*** trata a las llamadas de cola como un ***goto*** o salto de control (jump)
    - Se pueden hacer un ***numero indefinido*** de llamados de cola ***sin causar stackoverflow***
    - Es por esto que se recomienda usar recursión de cola en algoritmos que tienen mucho anidamiento

Recursión Simple:

```
(define factorial
  (lambda (n)
    (let fact ((i n))
      (if (= i 0)
          1
          (* i (fact (- i 1)))))))
```

n! = n * (n-1)!
0! = 1

Recursión de Cola:

```
(define factorial
  (lambda (n)
    (let fact ((i n) (a 1))
      (if (= i 0)
          a
          (fact (- i 1) (* a i))))))
```

n! = n * (n-1) * (n-2) * ... 2 * 1

Todas las operaciones se hacen en los argumentos del llamado recursivo

fib(n) = fib(n-1) + fib(n-2)
fib(0) = 0 y fib(1) = 1

Recursión Simple:

```
(define fibonacci
  (lambda (n)
    (let fib ((i n))
      (cond ((= i 0) 0)
            ((= i 1) 1)
            (else (+ (fib (- i 1)) (fib (- i 2))))))))
```

Recursión de Cola:

```
(define fibonacci1
  (lambda (n)
    (if (= n 0)
        0
        (let fib ((i n) (a1 1) (a0 0))
          (if (= i 1)
              a1
              (fib (- i 1) (+ a1 a0) a1))))))
```

Asignación

martes, 30 de noviembre de 2021 15:15

- ***Let*** permite ligar un valor a una (nueva) variable en su cuerpo (local), mientras que ***define*** permite ligar un valor a una (nueva) variable de nivel superior. Sin embargo ninguna de estas dos permite cambiar el ligado de una variable que ya existe, como lo haría una ***asignación***
- ***Set!*** permite re-ligar una variable existente a un nuevo valor, pero no establece un nuevo ligado, solo cambia uno existente para que evaluaciones posteriores se evalúen con el nuevo valor
- Son útiles para ***actualizar estados y crear nuevas estructuras***

```
(define abcde '(a b c d e))
; => abcde

abcde
; => (a b c d e)


(set! abcde (cdr abcde))
; => abcde

abcde
; => (b c d e)
```

```
;;; haga-stack: es un procedimiento que permite crear un stack
;;; que tiene las operaciones: vacio? , push! , pop! y tope!

(define haga-stack
  (lambda ()
    (let ((st '()))
      (lambda (op . args)
        (cond
          ((eqv? op 'vacio?)(null? st))
          ((eqv? op 'push!) (begin (set! st (cons (car args) st))) st)
          ((eqv? op 'pop!)  (begin (set! st (cdr st))) st)
          ((eqv? op 'tope!) (car st))
          (else             "operacion no valida")
         )))))
; => haga-stack
```

```
(define st (haga-stack))   => st

(st 'vacio?)               => #t

(st 'push! 'perro)         => (perro)

(st 'push! 'gato)          => (gato perro)

(st 'push! 'canario)       => (canario gato perro)

(st 'tope!)                => canario

(st 'vacio?)               => #f

(st 'pop!)                 => (gato perro)
```

Ligado de Variables

martes, 30 de noviembre de 2021 15:27

- **Expresión Lambda**
  - Permite crear procedimientos, cuyo cuerpo se evalúa secuencialmente
  - En el momento de la evaluación se ligan los parámetros formales a los actuales
  - Los parámetros formales se especifican de tres formas
    - Lista propia
    - Lista impropia
    - Variable única

```
((lambda (x y) (+ x y)) 3 4)          => 7

((lambda (x . y) (list x y)) 3 4)     => (3 (4))

((lambda x x) 3 4)                    => (3 4)
```

- **Ligado de Referencias a una Variable**
  - Es un error evaluar una referencia a una variable de nivel superior antes de definirla
  - No lo es que una referencia a una variable aparezca dentro de una expresión no evaluada

```
(define x 'a)                         => x
(list x x)                            => (a a)

(let ((x 'b)) (list x x))             => (b b)
```

```
(define f (lambda (x) (g x)))         => f
(define g (lambda (x) (+ x x)))       => g
(f 3)                                 => 6
```

- **Ligado Local**
  - ***Let***
    - Cada variable se liga a su valor correspondiente
    - Las expresiones de valor en la definición están del ámbito de las variables
    - Se recomienda su uso para valores independientes, donde no importa el orden de evaluación
  - ***Let****
    - Se asegura que el orden de evaluación sea de izquierda a derecha
    - Recomendado si hay una dependencia lineal entre los valores o el orden de evaluación es importante

```
(let ((x 1) (y 2))
        (let ((x y) (y x))
                (list x y)))          => (2 1)
```

```
(let ((x 1) (y 2))
        (let* ((x y) (y x))
                (list x y)))          => (2 2)
```

  - ***Letrec***
    - Similar a ***let***, pero todos los valores están dentro del ámbito de todas las variables
    - Permite definición de procedimientos mutuamente recursivos
    - El orden de evaluación no es especificado
    - Se recomienda su uso si hay una ***dependencia circular*** entre las variables y sus valores y el orden no es importante
    - Definiciones son también visibles en los valores de las variables
    - Se usa principalmente para definir funciones lambda
    - Esta la ***restricción*** que cada valor debe ser evaluable sin la necesidad de evaluar otros valores definidos

```
;;; letrec hace visible las variables dentro de los valores definidos,
;;; permitiendo definiciones recursivas con ámbito local

(letrec ((suma (lambda (ls)
                 (if (null? ls)
                     0
                     (+ (car ls) (suma (cdr ls)))
                 )
         )
   ))
   (suma '(1 2 3 4 5 6))
)
```

```
(letrec ((suma (lambda (x)
                 (if (zero? x)
                     0
                     (+ x (suma (- x 1)))
                 )))))
(suma 10))

=> 55
```

```
(letrec ((f (lambda () (+ x 2)))
         (x 1))
 (f))
=> 3
```

¡Es válido!

```
(letrec ((y (+ x 2))
         (x 1))
 y)
=> error
```

¡No es válido

```
(letrec
  (
   (par? (lambda (x)
                 (or (= x 0)(impar? (- x 1))))
   )
   (impar? (lambda (x)
                 (and (not (= x 0))(par? (- x 1))))
   )
  )
  (list (par? 20) (impar? 20))
)
=> (#t #f)
```

Recursión Mutua

Ligado Local: **letrec** y **let** con nombre

- La expresión let con nombre:

```
((lambda(val)
   (let nombre ((var val)) exp1 exp2 ...)
```

- equivale a la expresión letrec:

```
((letrec ((nombre
      (lambda (var) exp1 exp2 ...)))
  nombre
 )
 val ...)
```

Ejemplo de let con nombre:
(comparar con ejercicio de ppt 61)

```
((lambda (ls)
  (let suma ((l ls))
          (if (null? l)
                  0
                  (+ (car l) (suma (cdr l)))
          )
    )
  )
  '(1 2 3 4 5 6)
)

=> 21
```

Otras Operaciones en Scheme

martes, 30 de noviembre de 2021 20:02

- **Igualdad - Equivalencia**
  - (eq? obj_1 obj_2)
    retorno: #t *si son idénticos*
  - (eqv? obj_1 obj_2)
    retorno: #t *si son equivalentes*
  - (equal? obj_1 obj_2)
    retorno: #t *si tienen la misma estructura y contenido*
  - eqv? es similar a eq?, salvo que no es dependiente de la implementación, pero es algo más costoso.
  - eq? no permite comparar en forma fiable números.
  - equal? es similar a eqv?, salvo que se aplica también para strings, pares y vectores.

```
(eq? 'a 'a)                    => #t
(eq? 3.1 3.1)                  => #f
(eq? (cons 'a 'b) (cons 'a 'b)) => #f

(eqv? 'a 'a)                   => #t
(eqv? 3.1 3.1)                 => #t
(eqv? (cons 'a 'b) (cons 'a 'b)) => #f

(equal? 'a 'a)                 => #t
(equal? 3.1 3.1)               => #t
(equal? (cons 'a 'b) (cons 'a 'b)) => #t
```

- **Listas Asociativas**
  - Es una lista propia cuyos elementos son ***pares*** que tienen la forma ***clave-valor***
  - Las asociaciones son útiles para almacenar información (*valor*) relacionada con un objeto (*clave*)
  - (assq *obj alist*)
    retorno: *primer elemento de alist cuyo* car *es equivalente a obj, sino* #f
  - (assv *obj alist*)
    ídem
  - (assoc obj *alist*)
    ídem

```
(define e '((a 1) (b 2) (c 3)))
```

  - (assq 'a e) ⇒ (a 1)
    (assq 'b e) ⇒ (b 2)
    (assq 'd e) ⇒ #f

- **Eval**

(eval *obj*)
retorno: evaluación de *obj* como programa Scheme

- *obj* debe ser un programa válido de Scheme.
- El ámbito actual no es visible a *obj*, comportándose éste como si estuviera en un nivel superior de otro ambiente.
- No pertenece al estándar de ANSI/IEEE.

```
(eval 3)              => 3
(eval '(+ 3 4))       => 7
(eval (list '+ 3 4))  => 7
```

ESTA SECCION ES BASICAMENTE UNA DOCUMENTACION DE FUNCIONES DE SCHEME, O SEA VER EL PDF