Lenguajes de Programación
Programación funcional

Roberto Bonvallet

Departamento de Informática
Universidad Técnica Federico Santa María

Concepto de función

En el paradigma imperativo:

- Una función es una secuencia de instrucciones
- El valor de retorno se obtiene a través de una serie de manipulaciones de estado
- Existencia de estado implica posibles efectos laterales
- Consecuencia de la arquitectura de von Neumann

Concepto de función

En matemáticas:

- Una función es un mapeo de los elementos de un conjunto a los de otro: $f : A \longrightarrow B$
- Generalmente descrita por una expresión o una tabla de valores
- Evaluación controlada por recursión y expresiones condicionales
- Libre de efectos laterales

Influencia del paradigma imperativo en la programación

- Lenguajes imperativos están diseñados para explotar la arquitectura del computador
- Apego a la arquitectura es una restricción innecesaria al proceso de desarrollo de software
- Lenguajes diseñados según otros paradigmas son impopulares debido a penalizaciones de rendimiento
- Máquina de Turing: modelo matemático para estudiar capacidades y limitaciones del computador

Fundamentos de los lenguajes funcionales

- Objetivo: emular las funciones matemáticas lo más posible.
- No se usa variables ni asignaciones.
- Repetición a través de recursión.
- Transparencia referencial.
- Funciones son objetos de primera clase.
- Un programa consiste de definiciones de funciones y aplicaciones de ellas.

Funciones

Funciones simples

- Ejemplo: cubo$(x) \equiv x * x * x$
- x representa cualquier valor del dominio, pero está fijo en un elemento específico durante la evaluación
- Durante la evaluación, cada aparición de un parámetro está ligada a un valor y es considerada constante
- Notación lambda para funciones anónimas:

$$\lambda(x)\, x * x * x$$

- Evaluación de una función anónima:

$$\big(\lambda(x)\, x * x * x\big)(2) = 8$$

Funciones

Formas funcionales (funciones de mayor orden)

- Reciben funciones como parámetros, o entregan funciones como resultado. Por ejemplo:

  composición

  $$\big(f \circ g\big)(x) = f\big(g(x)\big)$$

  construcción

  $$\big[g, h, i\big](x) = \big(g(x), h(x), i(x)\big)$$

  aplicar a todo

  $$\alpha\big(j, (x, y, z)\big) = \big(j(x), j(y), j(z)\big)$$

Comparación entre un programa imperativo y uno funcional

- Problema sencillo: aplicar dos funciones a una lista de valores
- Programa imperativo:

target $\leftarrow$ []
for i **in** source **do**
  $t_1 \leftarrow g(i)$
  $t_2 \leftarrow f(t_1)$
  target.append(t_2)
end for

- Distintas versiones funcionales del mismo programa:

target $\leftarrow \alpha(f \circ g, \text{source})$
target $\leftarrow \alpha\Big(\lambda(x)\, f\big(g(x)\big), \text{source}\Big)$
target $\leftarrow \Big[f\big(g(i)\big)\ \forall i \in \text{source}\Big]$

Introducción a Scheme

Características:

- dialecto de LISP
- sintaxis y semántica simples
- nombres tienen ámbito estático
- no es puramente funcional

Funcionamiento del intérprete de Scheme

- Entorno interactivo implementa un ciclo lectura-evaluación-escritura
- Se puede cargar las expresiones desde un archivo para facilitar el desarrollo.
- Las llamadas a funciones se evalúan así:
  1. se evalúa los parámetros;
  2. se aplica la función sobre los parámetros.

Funcionamiento del intérprete de Scheme

Ejemplo de sesión interactiva

```
12
--> 12
'(1 2 3 4)
--> (1 2 3 4)
(+ 1 2 3)
--> 6
(string-append "hola " "mundo")
--> "hola mundo"
```

Sintaxis del lenguaje

Scheme utiliza la misma sintaxis para:

1. listas:

```
(1 2 3 4 5)
```

2. llamadas a función:

```
(max 43 -23 15 58 21)
```

3. formas sintácticas:

```
(if (< x 0) x (- x))
```

Constantes literales

- Valores literales pueden ser:
  - números enteros, reales, racionales y complejos:

```
123456789
3.14159
22/7
+27.0-3.0i
5.0@3.0
```

  - strings:

```
"Hola mundo"
```

  - caracteres:

```
#\a
#\space
```

  - booleanos:

```
#t
#f
```

Listas

- Notación de listas:

```
(a b c d)
```

- Los elementos pueden ser de cualquier tipo:

```
(1.0 "Hola mundo" 3)
```

- Las listas se pueden anidar:

```
(a (b c) (d e (f) g))
```

- Las listas se tratan como objetos inmutables

Representación interna de las listas

- implementadas como listas enlazadas
- los nodos se llaman *pares* o *celdas cons*
- el final de una lista se marca con la lista vacía

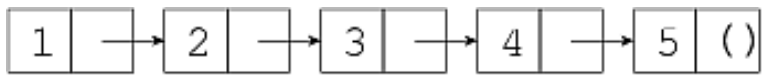

Figura: Representación de (1 2 3 4 5)

Representación interna de las listas

- implementadas como listas enlazadas
- los nodos se llaman *pares* o *celdas cons*
- el final de una lista se marca con la lista vacía

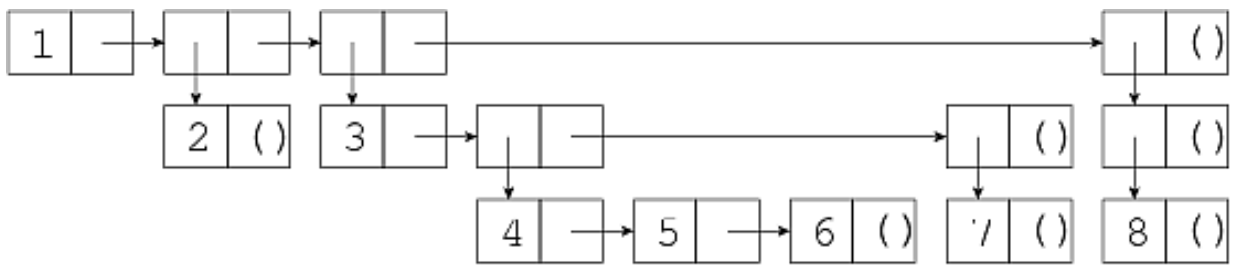

Figura: Representación de (1 (2) (3 (4 5 6) (7)) ((8)))

Celdas cons

- Las celdas cons son el tipo de datos estructurado fundamental de Scheme.
- El campo izquierdo de una celda se denomina el *car*, y el campo derecho, el *cdr*.
- En una lista enlazada propia, las celdas tienen:
  - un valor en el car;
  - una referencia a una lista en el cdr.

Celdas cons

- Las funciones car y cdr obtienen el car y el cdr de una celda.
- La función **cons** permite crear una celda cons definiendo su car y su cdr, y por lo tanto se puede usar para crear listas enlazadas.

```
(car '(1 2 3 4 5))
--> 1
(cdr '(1 2 3 4 5))
--> (2 3 4 5)
(cons 'a '(b c d))
--> (a b c d)
```

Listas impropias

- Una lista es impropia cuando el cdr de su última celda no es la lista vacía.
- Las listas impropias se representan usando la notación de par con punto.

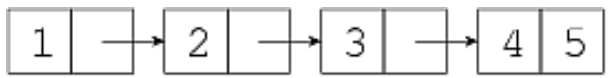

Figura: Representación de (1 2 3 4 . 5)

Listas impropias

- Una lista es impropia cuando el cdr de su última celda no es la lista vacía.
- Las listas impropias se representan usando la notación de par con punto.

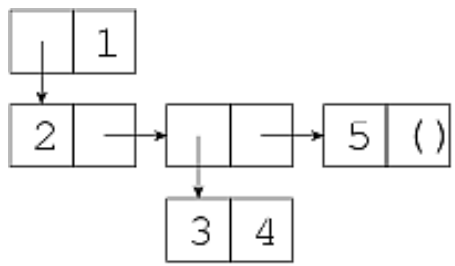

Figura: Representación de ((2 (3 . 4) 5) . 1)

Listas impropias

- Una lista es impropia cuando el cdr de su última celda no es la lista vacía.
- Las listas impropias se representan usando la notación de par con punto.

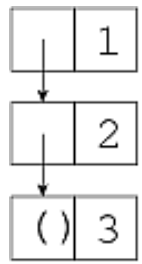

Figura: Representación de ((() . 3) . 2) . 1)

Llamadas a funciones

- Llamadas a funciones utilizan la misma notación que las listas:

  $(\texttt{funcion}\ \texttt{arg}_1\ \texttt{arg}_2\ \ldots\ \texttt{arg}_N)$

- Para evitar que una lista sea evaluada como función, se utiliza el operador de citado:

```
(a b c d)
error: 'a' no es una funcion
(quote (a b c d))
--> (a b c d)
'(a b c d)
--> (a b c d)
```

Funciones aritméticas

- Scheme provee las funciones aritméticas elementales $+$, $-$, $*$ y $/$:

```
(+ 2 -3)
--> -1
(* 1 2 3 4 5)
--> 120
(+ (* 3 3) (* 4 4))
--> 25
(- (/ 81 9) (*) (+))
--> 8
```

- Notación prefija evita ambigüedad en la evaluación.

Funciones anónimas

- La forma sintáctica lambda permite definir funciones anónimas.
- Sintaxis:

  (lambda (arg$_1$ arg$_2$ ...) expr$_1$ expr$_2$ ...)

- Ejemplo:

```
(lambda (x y) (+ (* x x) (* y y)))
--> #<procedure>
((lambda (x y) (+ (* x x) (* y y))) 3 4)
--> 25
```

Sintaxis alternativas de lambda

- Guarda en args la lista de todos los argumentos:

  $(\text{lambda}\ \ \text{args}\ \ \text{expr}_1\ \ \text{expr}_2\ \ \ldots)$

- Guarda en rest todos los argumentos adicionales:

  $(\text{lambda}\ \ (\text{arg}_1\ \ \ldots\ \ \text{arg}_N\ \ .\ \ \textbf{rest})\ \ \text{expr}_1\ \ \text{expr}_2\ \ \ldots)$

Ligados locales

- La forma sintáctica **let** establece ligados de nombres locales.
- Nombres ligados con **let** son visibles sólo en el cuerpo del **let**.
- Sintaxis:

```
(let ((var1 val1) (var2 val2) ...)
     expr1 expr2 ...)
```

- Ejemplo:

```
(let ((a 17) (b 31))
     (+ (* 1/2 a b) (* 1/3 a b)))
--> 439.166666666667
((lambda (x y)
     (let ((x-squared (* x x))
           (y-squared (* y y)))
          (sqrt (+ x-squared y-squared))))
5 12)
--> 13.0
```

Definiciones de nivel superior

- La forma sintáctica define liga un nombre a una expresión.
- Sintaxis:

```
(define nombre expr)
```

- Ejemplo:

```
(define n 25)
(define square (lambda (x) (* x x)))
(square n)
--> 625
```

Expresiones condicionales simples

- La forma sintáctica if permite evaluar una expresión de manera condicional.
- Sintaxis:

```
(if condicion consecuencia alternativa)
```

- Ejemplo:

```
(define signo
  (lambda (x)
    (if (>= x 0) 'positivo 'negativo)))
(signo -2)
--> negativo
(signo 5)
--> positivo
```

Expresiones condicionales múltiples

- La forma sintáctica **cond** permite evaluar una expresión usando distintos casos.
- Sintaxis:

  (**cond** clausula$_1$ clausula$_2$ ...)

- Cada cláusula tiene alguna de estas formas:
  1. (condicion)
  2. (condicion expr$_1$ expr$_2$ ...)
  3. (condicion => expr)
- La última cláusula puede tener la forma:
  4. (else expr$_1$ expr$_2$ ...)

Ejemplo de uso de cond

```
(define income-tax
  (lambda (income)
    (cond
      ((<= income 10000) (* income .05))
      ((<= income 20000)
        (+ (* (- income 10000) .08)  500.00))
      ((<= income 30000)
        (+ (* (- income 20000) .13) 1300.00))
      (else
        (+ (* (- income 30000) .21) 2600.00))
)))
```

Evaluación de valores de verdad

- Todos los objetos son considerados verdaderos, excepto #f.
- Existen las formas sintácticas **and**, **or** y **not**.
- **and** y **or** hacen cortocircuito.
- En general, los predicados lógicos tienen nombres terminados en ?.

Ejemplos de valores de verdad

```
(if 1 'true 'false)
--> true
(if '() 'true 'false)
--> true
(if #f 'true 'false)
--> false
(not #t)
--> #f
(not "false")
--> #f
(or)
--> #f
(or #f #t)
--> #t
(or #f 'a #t)
--> 'a
```

Ejemplos de predicados lógicos

```
(< 1 2 3 4 5)
--> #t
(null? '())
--> #t
(null? '(1 2 3))
--> #f
(pair? '(8 . 2))
--> #t
(eqv? 3 2)
--> #f
(number? 15)
--> #t
(list? "Veamos si soy una lista ...")
--> #f
```

Recursión simple

- En Scheme, la iteración se hace usando recursión.
- La recursión es más general que construcciones tipo *for* o *while*, y elimina la necesidad de usar variables de control.
- Una función recursiva debe tener:
  1. un caso base, y
  2. un paso de recursión.
- Ejemplo:

```
(define length
    (lambda (ls)
        (if (null? ls)
            0
            (+ 1 (length (cdr ls))))))
(length '())
--> 0
(length '(a b c))
--> 3
```