Programación Funcional

Lucas Fanchin | Bruno Motto

Identificadores

- **Formación**:
  - Letras y números: `[a-z] [A-Z] [0-9]`
  - Símbolos: `? ! . + - * / < = > : $ % & _ ~ @ ◦`
- **Ejemplos**: `Hola`, `n`, `x`, `x3`, `?$&*!!`
- **Delimitadores**:
  - Espacio en blanco
  - Comillas dobles (`"`)
  - Paréntesis
  - Carácter de comentario (`;`)
- **Características**:
  - No hay límite de longitud.
  - No es case-sensitive (ej. `Abc`, `abc`, `aBc` son el mismo identificador).

Tipos de Datos

- *Estructuras y Listas*
  - Formato: Se encierran entre paréntesis.
  - Ejemplo: `(a b c)`
- *Valores Booleanos*
  - **Verdadero**: `#t`
  - **Falso**: `#f`
- *Strings*
  - **Formato**: Encerrados en comillas dobles (`"`).
  - **Ejemplo**: `"TecPro"`
- *Caracteres*
  - **Formato**: Precedidos por `#\`.
  - **Ejemplo**: `#\a`
- *Conversión de Tipos*
  - **Formato**: `tipo1->tipo2`.
  - **Ejemplo**: `vector->list`

Quote (')

Uso: Trata un identificador como símbolo y no como variable.

Predicados

- **Características**: Finalizan en `?` y retornan `#t` o `#f`.
    - **Ejemplos**: `eq?`, `zero?`, `string=?`

Funciones Básicas para Condiciones

- **`number?`**: Retorna `#t` si el argumento es un número.
- **`boolean?`**: Retorna `#t` si el argumento es un booleano.
- **`list?`**: Retorna `#t` si el argumento es una lista.
- **`null?`**: Retorna `#t` si el argumento es una lista vacía.
- **`symbol?`**: Retorna `#t` si el argumento es un símbolo.
- **`char?`**: Retorna `#t` si el argumento es un carácter.
- **`string?`**: Retorna `#t` si el argumento es una cadena.
- **`eq?`**: Retorna `#t` si ambos argumentos son el mismo objeto en memoria.
- **`eqv?`**: Retorna `#t` si ambos argumentos son equivalentes.
- **`pair?:`**Retorna `#t` si el argumento es un par cons (una lista no vacía)

Operadores de Listas

- **`car`**: Retorna el primer elemento de la lista.
- **`cdr`**: Retorna el resto de la lista (todo menos el primer elemento).
- **`cons`**: Construye listas. Recibe dos argumentos y crea una nueva lista.
    - **Ejemplo**: `(cons 1 '(2 3))` retorna `(1 2 3)`.
- **`cadr`**: Accede al segundo elemento de una lista. (no visto pero útil).

Asignaciones

- **`let`**: Define variables locales.
    - **Ejemplo**: `(let ([x 5] [y 6]) (+ x y))`
- **`let*`**: Permite asignaciones secuenciales, donde las definiciones internas pueden ver a las variables externas.
    - **Ejemplo**: `(let* ([x 5] [y (+ x 1)]) y)` resulta en `6`.
- **`letrec`**: Permite definir un conjunto de pares variable-valor y sentencias que las referencian. Las variables son visibles en la cabecera.
    - **Ejemplo**: `(letrec ([even? (lambda (n) (or (zero? n) (odd? (- n 1))))] [odd? (lambda (n) (and (not (zero? n)) (even? (- n 1))))]) (even? 4))` retorna `#t`.

Funciones Lambda

- **Definición**: Una función anónima que puede ser utilizada para definir funciones pequeñas y rápidas sin necesidad de nombrarlas.
- **Sintaxis**: `(lambda (arg1 arg2 ... argn) (cuerpo))`
- **Uso**:
  - Para crear funciones rápidas y específicas en el lugar donde se necesitan, sin tener que definirlas por nombre.
- **Ejemplos**:
  - *Definición básica:* `(lambda (x) (* x x))` define una función que toma un argumento y devuelve su cuadrado.
  - *Definir una función local:* `(let ([cuadrado (lambda (x) (* x x))]) (cuadrado 5))` retorna `25`.
  - *Dentro de* **`map`**: `(map (lambda (x) (* x x)) '(1 2 3 4))` retorna `(1 4 9 16)`.

Recursividad

- **Definición**: Ocurre cuando un procedimiento se llama a sí mismo.
  - **Ejemplo**: `(letrec ([factorial (lambda (n) (if (zero? n) 1 (* n (factorial (- n 1)))))]) (factorial 5))` retorna `120`.

Vectores

- **Definición**: Secuencia de objetos precedidos por un `#` o con la sintaxis `(vector v1 v2 ... vn)`.
- **Funciones para Vectores**:
  - `(make-vector n)` o `(make-vector n obj)`: Retorna un vector de `n` posiciones.
  - `(vector-length vector)`: Retorna la cantidad de elementos de un vector.
  - `(vector-ref vector n)`: Retorna la enésima posición de un vector.
  - `(vector-set! vector n obj)`: Establece el valor de la enésima posición del vector como `obj`.
  - `(vector-fill! vector obj)`: Reemplaza cada elemento del vector por `obj`.
  - `(vector->list vector)`: Devuelve una lista a partir de un vector.
  - `(list->vector list)`: Convierte una lista en vector.

Mapeo de Procedimientos a Listas

- **`map`**: Aplica el procedimiento a cada elemento de la lista y devuelve una lista con los resultados. Es posible tener múltiples argumentos.

- **`for-each`**: aplica el procedimiento a cada elemento de la lista pero devuelve un <void>

Estructuras

`define-struct`: permite crear una estructura con los campos que se indican. Se crean tres operaciones:

- un constructor: **`make-<nom-struct>`**
- métodos selectores: **`<nom-struct>-<campo>`**
- métodos accesores: **`set-<nom-struct>-<campo>!`**

Condicionales

- **`if`**: Es una expresión condicional que evalúa una condición y devuelve un valor dependiendo de si la condición es verdadera o falsa.
    - Ejemplo:
```
(if (condición)
    resultado-si-verdadero
    resultado-si-falso)
```
- **`cond`**: Es una forma de expresión condicional más general que `if`, que permite evaluar múltiples condiciones secuencialmente y devolver un valor basado en la primera condición que se cumpla.
    - Ejemplo:
```
(cond
    (condición-1 resultado-1)
    (condición-2 resultado-2)
    …….
    (else resultado-por-defecto))
```