



Juan fernando Muñoz Cabrera

Proyecto 1 - Fase 1

Lisp

Lisp es un lenguaje de programación de propósito general que fue desarrollado en 1958 por John McCarthy en el MIT (Massachusetts Institute of Technology). El nombre "Lisp" proviene de "LIST Processing" debido a su énfasis en la manipulación de listas como estructuras de datos fundamentales. Es uno de los lenguajes de programación más antiguos que aún se usan en la actualidad y que ha influido significativamente en el diseño de otros lenguajes. (McCarthy, n.d.)

Lisp fue desarrollado originalmente como un lenguaje para la inteligencia artificial (IA) y la investigación en el MIT.

A lo largo de los años, han surgido varias versiones y extensiones, como Common Lisp, Scheme y Clojure, entre otros.

Características de Lisp:

- **Sintaxis Simple y Consistente:** Este se basa en la notación de paréntesis y utiliza una sintaxis uniforme basada en listas. (McCarthy, n.d.)
- **Dinámicamente Tipado:** No se requiere especificar tipos de datos de manera explícita. (McCarthy, n.d.)
- **Programación Funcional:** Soporta paradigmas de programación funcional, lo que significa que las funciones son ciudadanos de primera clase y se pueden pasar como argumentos y retornar como valores. (McCarthy, n.d.)
- **Gestión de Memoria Automática:** La mayoría de las implementaciones de Lisp tienen un recolector de basura que gestiona automáticamente la memoria.
- **Potente Sistema de Macro:** Permite a los programadores extender el lenguaje escribiendo código que se ejecuta durante la compilación. (McCarthy, n.d.)

Hoy en día, se utiliza Lisp en aplicaciones de IA, sistemas expertos, procesamiento de datos y otras áreas donde la flexibilidad y la capacidad de manipulación de datos son importantes.

Programación Funcional vs Programación Orientada a Objetos:

Programación Funcional: Se centra en la evaluación de funciones y el tratamiento de los datos como inmutables. (*Es Lo Mismo POO, Que Programación Funcional?*, n.d.)

Programación Orientada a Objetos (POO): Se centra en la manipulación de objetos que encapsulan datos y comportamiento. Los objetos interactúan entre sí a través de métodos y mensajes. (*Es Lo Mismo POO, Que Programación Funcional?*, n.d.)

Java Collections Framework

El Java Collections Framework (JCF) es una biblioteca en el lenguaje de programación Java que proporciona una arquitectura unificada para representar y manipular colecciones de objetos. Esta biblioteca está diseñada para facilitar la manipulación y almacenamiento de datos de manera eficiente y flexible.

Jerarquía de Interfaces del JCF:

Collection: La interfaz raíz de la jerarquía de colecciones. Define las operaciones básicas que se aplican a todas las colecciones, cómo agregar elementos, eliminar elementos, verificar si un elemento está presente, entre otras.

List: Extiende la interfaz Collection y representa una colección ordenada de elementos donde se permite duplicados.

Set: También extiende Collection, pero representa una colección que no permite duplicados. Las implementaciones comunes incluyen HashSet, TreeSet y LinkedHashSet.

Queue: Extiende Collection y modela una estructura de datos FIFO (primero en entrar, primero en salir). Las implementaciones incluyen LinkedList y PriorityQueue.

Map: No extiende Collection, pero representa una colección de pares clave-valor donde cada clave está asociada a un único valor. Las implementaciones incluyen HashMap, TreeMap, LinkedHashMap, entre otras.

Implementaciones del JCF:

ArrayList: Implementa la interfaz List utilizando un array dinámico para almacenar los elementos. Es eficiente para el acceso aleatorio, pero puede ser costoso para insertar y eliminar elementos en posiciones intermedias.

LinkedList: Implementa la interfaz List utilizando una lista doblemente enlazada. Es eficiente para insertar y eliminar elementos en posiciones intermedias, pero menos eficiente para el acceso aleatorio.

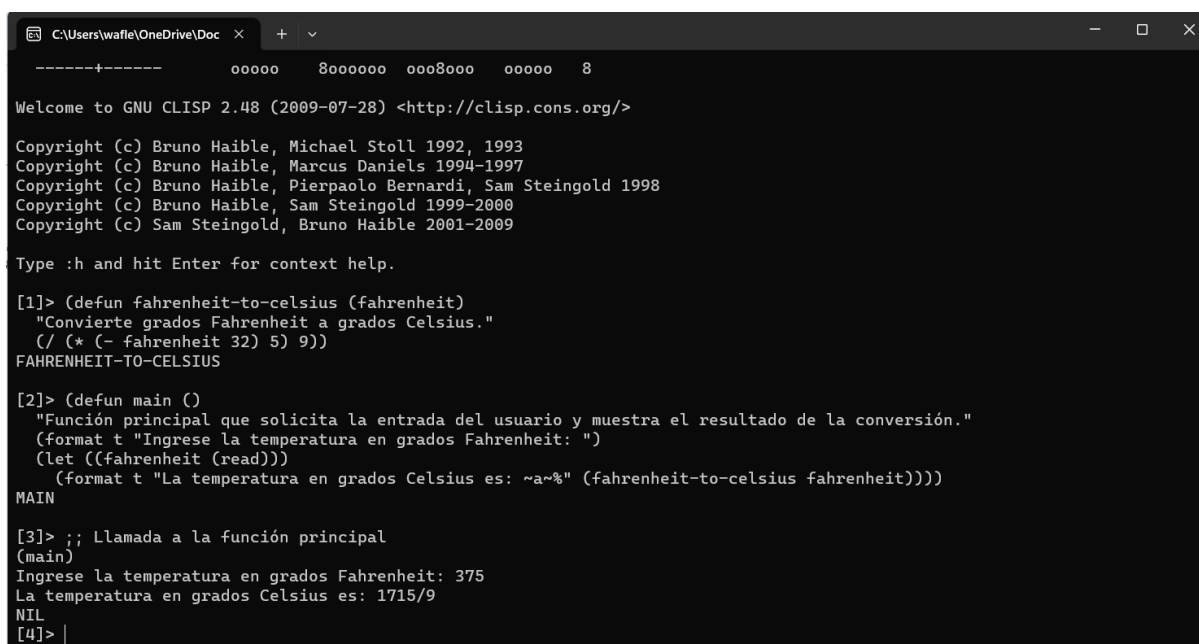
HashSet: Implementa la interfaz Set utilizando una tabla hash para almacenar los elementos. Ofrece un rendimiento constante para operaciones básicas como agregar, eliminar y buscar elementos.

TreeSet: Implementa la interfaz Set utilizando un árbol binario de búsqueda para almacenar los elementos. Los elementos se almacenan ordenados según su valor natural o utilizando un comparador proporcionado por el usuario.

HashMap: Implementa la interfaz Map utilizando una tabla hash para almacenar los pares clave-valor.

TreeMap: Implementa la interfaz Map utilizando un árbol rojo-negro para almacenar los pares clave-valor.

LinkedHashMap: Implementa la interfaz Map utilizando una lista doblemente enlazada y una tabla hash.



```
C:\Users\waffle\OneDrive\Doc x + v
-----+-----      00000      8000000      0008000      00000      8

Welcome to GNU CLISP 2.48 (2009-07-28) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2009

Type :h and hit Enter for context help.

[1]> (defun fahrenheit-to-celsius (fahrenheit)
      "Convierte grados Fahrenheit a grados Celsius."
      (/ (* (- fahrenheit 32) 5) 9))
FAHRENHEIT-TO-CELSIUS

[2]> (defun main ()
      "Función principal que solicita la entrada del usuario y muestra el resultado de la conversión."
      (format t "Ingrese la temperatura en grados Fahrenheit: ")
      (let ((fahrenheit (read)))
        (format t "La temperatura en grados Celsius es: ~a~%" (fahrenheit-to-celsius fahrenheit))))
MAIN

[3]> ;; Llamada a la función principal
(main)
Ingrese la temperatura en grados Fahrenheit: 375
La temperatura en grados Celsius es: 1715/9
NIL
[4]> |
```

```
C:\Users\waffle\OneDrive\Doc x + v
[1]> (defun fibonacci (n)
  "Calcula el término n de la serie de Fibonacci."
  (if (or (= n 0) (= n 1))
      n
      (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
FIBONACCI

[2]> (defun factorial (n)
  "Calcula el factorial de un número dado."
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
FACTORIAL

[3]> (defun main ()
  "Función principal que solicita al usuario un número y calcula su término Fibonacci y factorial."
  (format t "Ingrese un número para calcular su término Fibonacci y factorial: ")
  (let ((numero (read)))
    (format t "El término ~a de la serie de Fibonacci es: ~a~%" numero (fibonacci numero))
    (format t "El factorial de ~a es: ~a~%" numero (factorial numero))))
MAIN

[4]> ;; Llamada a la función principal
(main)
Ingrese un número para calcular su término Fibonacci y factorial: 7
El término 7 de la serie de Fibonacci es: 13
El factorial de 7 es: 5040
NIL
[5]> |
```

UML

Lispcode
<ul style="list-style-type: none">- suma(a: int, b: int)- resta(a: int, b: int)- multiplicacion(a: int, b: int)- division(a: int, b: int)- defun-lisp(nombre: string, parametros: list, cuerpo: list)- setq-lisp(nombre: string, valor: int)- atom-lisp(x: list)- list-lisp(x: list)- equal-lisp(a: int, b: int)- menor-lisp(a: int, b: int)- mayor-lisp(a: int, b: int)- cond-lisp(clauses: list)- eval-lisp(exp: list)