

AF6, Reporte que describa la ejecución de un programa funcional.



UANL
UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN



FIME
FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

Lenguajes de Programación.

Ing. Karla Patricia Uribe Sierra.

Grupo 005.

M5.

Semestre enero – junio 2025.

Matthew Alejandro Martínez Zambrana.

2223170.

IAS.

San Nicolás de los Garza, N.L, viernes 21 de marzo de 2025.

Scheme.

Es un lenguaje de programación funcional y uno de los dialectos más conocidos del lenguaje Lisp. Fue desarrollado en los años 70 por Guy L. Steele y Gerald Jay Sussman en el MIT. Scheme es conocido por su simplicidad y elegancia, así como su flexibilidad y poder. Es un lenguaje de propósito general que permite la programación funcional, la programación orientada a objetos y la programación imperativa.

Los primeros trabajos se centraron en la exploración de la semántica de programación y la implementación de intérpretes eficientes. La primera versión del lenguaje, conocida como MIT Scheme, se publicó en 1975.

Scheme ha evolucionado y ha sido influenciado por las investigaciones y desarrollos en teorías de lenguajes de programación. La versión estandarizada más reciente de Scheme es R7RS, publicada en 2013.

Características.

Simplicidad y elegancia. Scheme tiene una sintaxis sencilla y uniforme. Utiliza una notación de prefijo y paréntesis para todas las expresiones, lo que simplifica la sintaxis y facilita la manipulación del código como datos.

Closures. Significa que las funciones pueden capturar y referir su entorno de ejecución de funciones anidadas y de orden superior.

Macro Sistema. Potente sistema de macros que permite a los programadores extender el lenguaje y crear nuevos constructos de control.

Portabilidad. Altamente portable y hay múltiples implementaciones disponibles en diferentes plataformas, lo que lo hace adecuado para una amplia gama de aplicaciones.

Ejemplos de código en Scheme.

Ejemplo 1: Función de Factorial.

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1))))
```

Este ejemplo muestra una implementación recursiva de la función factorial en Scheme. La función toma un entero y devuelve el producto de todos los enteros posteriores menores iguales a ese número.

Ejemplo 2: Uso de cierres (closures).

```
(define (make-counter)
  (let ((count 0))
    (lambda () (set! count (+ count 1)))))
```

Este ejemplo muestra cómo crear una función que mantiene un estado interno utilizando cierres. La función make-counter devuelve una función que incrementa y devuelve un conteo cada vez que se llama.

hasta telecomunicaciones y procesamiento de datos en grandes volúmenes.

Ejemplos de lenguajes funcionales.

1. Haskell.

Un lenguaje funcional poco conocido por su fuerte tipificación estática y su capacidad de evaluación perezosa. Es ampliamente utilizado en la academia y la investigación.

Ejemplo: Implementación de la función de Fibonacci

Fibonacci :: Int → Int

$$\text{Fibonacci } 0 = 0$$

$$\text{Fibonacci } 1 = 1$$

$$\text{Fibonacci } n = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

2. Lisp:

Uno de los lenguajes de programación más antiguos, conocido por su flexibilidad y poder de metaprogramación. Lisp ha influido en muchos otros lenguajes de programación.

Ejemplo: Definición de una función simple

```
(defun factorial (n)
```

```
  (if (<= n 1)
```

```
    1
```

```
    (* n (factorial (- n 1)))))
```

3. Erlang.

Un lenguaje funcional diseñado para sistemas concurrentes y de alta disponibilidad. Es popular en el desarrollo de sistemas de telecomunicaciones y aplicaciones de mensajería.

Ejemplo: Función para enviar un mensaje en un sistema concurrente.

```
send_message(Recipient, Message) ->  
    Recipient ! {self(), Message}.
```

4. F#. (Fsharp).

Un lenguaje funcional que forma parte del ecosistema .NET. Combina características funcionales con capacidades imperativas y orientadas a objetos.

Ejemplo: Filtrado de una lista.

```
let evenNumbers = list.filter(fun x->x%2=0)[1;2;3;  
4;5;6]
```

5. Scala.

Un lenguaje que integra paradigmas funcionales y orientados a objetos. Es ampliamente utilizado en el desarrollo de aplicaciones de alto rendimiento y sistemas distribuidos.

Ejemplo: Uso de funciones anónimas.

```
val numbers = list(1,2,3,4,5,6)
```

```
val evenNumbers = numbers.filter(_%2 == 0)
```

Diagramas de Flujo de Recursión, Pilas y Listas.

La recursión es una técnica en la que una función se llama a sí misma para resolver un problema.

Ventajas.

- ✓ Hace que el código sea más elegante y fácil de entender.
- ✓ Útil para problemas que tiene una estructura recursiva natural (árboles, grafos).

Desventajas.

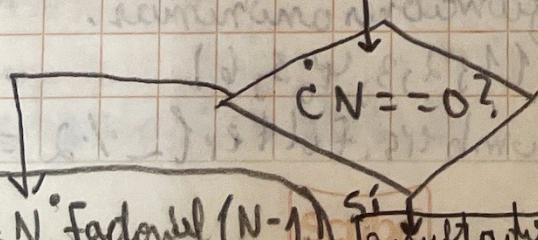
- ✗ Uso más memoria (se almacena cada llamada en la pila de ejecución).
- ✗ Puede causar desbordamiento de pila si no tiene un caso base adecuado.

Aplicaciones.

- Cálculo de factorial, fibonacci, potencias.
- Recorridos en estructuras de datos como árboles y grafos.
- Resolución de problemas como torres de Hanói y búsquedas binaria.

Ejemplo de diagrama de flujo

Resultado = factorial (N)



• Si n es 0, se retorna 1 (caso base).

• Si n es mayor

que 0, se multiplica n por factorial(n-1) que es otra llamada a la misma función.

Resultado = N * factorial (N-1)

Sí
Resultado = 1
Fin

- La función se sigue llamando a sí misma hasta llegar a 0!, que es 1.

- Luego, las llamadas regresan, multiplicando los valores acumulados.

Pila (Stack)

Una pila es una estructura de datos que sigue el principio LIFO (Last In, First Out) donde el último elemento agregado es el primero en salir.

Características

Push: Agrega un elemento a la pila.

Pop: Elimina el último elemento agregado.

Top: Muestra el elemento en la cima sin eliminarlo.

Estructura LIFO: Se apilan los elementos y se sacan en orden inverso.

Ejemplo en Java Script.

```
import java.util.Stack; // Importación necesaria para usar Stack

public class StackExample {
    public static void main (String [] args) {
        Stack < Integer > stack = new Stack < > ();
        // Agregar elementos a la pila.
        stack.push(10);
        stack.push(20);
        stack.push(30);
```

// Mostrar la pila antes de hacer pop

System.out.println("Pila actual: " + stack);

// Eliminar el elemento superior de la pila

System.out.println("Elemento eliminado: " + stack.pop());

// Mostrar la pila después de hacer pop

System.out.println("Pila después de pop: " + stack);

}

}

Pila actual: [10, 20, 30]

Elemento eliminado: 30

Pila después de pop: [10, 20]

Listas.

Una lista es una colección de elementos que pueden almacenarse de forma contigua (array) o dinámica (listas enlazadas).

Tipos de Listas:

Listas Enlazadas Simple: cada nodo tiene un puntero al siguiente nodo.

Listas Dblemente Enlazada: cada nodo tiene punteros al siguiente y al anterior.

Lista Circular: el último nodo apunta al primer nodo.

Características.

- ✓ Pueden crecer dinámicamente sin un tamaño fijo.
- ✓ Se pueden insertar y eliminar elementos sin mover todos los demás.
- ✓ Se accede a los elementos recorriendolos uno por uno.

Ejemplo en Python.

class Nodo:

```
def __init__(self, dato):  
    self.dato = dato  
    self.siguiente = None.
```

class ListaEnlazada:

```
def __init__(self):  
    self.cabeza = None.
```

```
def insertar_nodo(self, dato):  
    nuevo_nodo = Nodo(dato)  
    nuevo_nodo.siguiente = self.cabeza  
    self.cabeza = nuevo_nodo.
```

```
- def mostrar_lista(self):  
    actual = self.cabeza  
    while actual:  
        print(actual.dato, end=" → ")  
        actual = actual.siguiente  
    print("None")
```

Uso de la lista enlazada

lista = listaEnlazada()

lista.insertar_Enlazado(1)

lista.insertar_Enlazado(2)

lista.insertar_Enlazado(3)

lista.mostrar = lista(1) Salida: 3 → 2 → 1 → None

Terminal

PS C:\Users\angel\OneDrive - Universidad Autónoma
de Nuevo León.

3 → 2 → 1 → None.

atob = lista → lista → fib
atob = atob.flb
atob = atob.flb

(fib) → lista → fib
atob = atob.flb

atob.flb = atob.flb → fib
atob.flb = atob.flb → fib
atob.flb = atob.flb → fib

(fib) → fib → fib
atob.flb = atob.flb → fib

"←" = fib, atob, fib
atob.flb = fib
atob.flb = fib

Haskell

Haskell es un lenguaje de programación funcional puro y de propósito general, conocido por su fuerte sistema de tipos estáticos y su capacidad para realizar evaluaciones perezosas. Fue nombrado en honor al matemático y lógico Haskell Curry. Haskell es ampliamente utilizado en la academia y la investigación, y también ha encontrado aplicaciones en la industria, especialmente en el desarrollo de software crítico y de alta fiabilidad.

Historia y evolución.

Haskell fue desarrollado a finales de los años 80 y principios de los 90 como un esfuerzo colaborativo entre académicos e investigadores interesados en la programación funcional. La primera versión del lenguaje, Haskell 1.0, se publicó en 1990. Desde entonces, Haskell ha evolucionado y se ha mejorado continuamente, con la versión más reciente, Haskell 2010, publicada en 2010.

Características de Haskell.

- Funcional Puro.

Haskell es un lenguaje funcional puro, lo que significa que todas las funciones son funciones pures sin efectos secundarios. Esto facilita el razonamiento sobre el código y garantiza la predictibilidad del comportamiento del programa.

2. Evaluación Perezosa:

La evaluación perezosa, también conocida como evaluación diferida, permite que las expresiones no se evalúen hasta que su valor sea necesario. Esto puede mejorar la eficiencia y permitir la creación de estructuras de datos infinitas.

3. Sistema de Tipos Estático y Fuerte:

Haskell utiliza un sistema de tipos estático y fuerte que permite detectar errores de tipo en tiempo de compilación. Además, emplea inferencia de tipos, lo que significa que los tipos pueden deducirse automáticamente sin necesidad de ser explícitamente especificados.

4. Polimorfismo:

Haskell admite polimorfismo paramétrico, lo que permite escribir funciones y tipos genéricos que pueden tratar con múltiples tipos de datos.

5. Composición de funciones:

La composición de funciones es una característica fundamental de Haskell, permitiendo construir nuevas funciones a partir de otras más simples.

Propósito de Haskell:

Proporcionar un lenguaje de programación funcional puro y expresivo que permite la creación de software seguro, eficiente y mantenible. Algunos de los objetivos y ventajas de Haskell son:

- ✓ **Seguridad y consistencia:** Haskell facilita la escritura de programas correctos y seguros gracias a su sistema de tipos fuerte y a la inmutabilidad de los datos.
- ✓ **Claridad y concisión:** Permite crear y expresar algoritmos complejos de manera concisa y clara, reduciendo la cantidad de código necesario y facilitando su comprensión.
- ✓ **Aplicación concurrentes y Paralelas:** Es adecuado para la programación concurrente y paralela, gracias a su enfoque en la inmutabilidad y a las facilidades para manejar la concurrencia mediante monadas y otros constructos.

Ejemplos de código:

Ejemplo 1: Función de Factorial. Ejemplo 2: Filtrado de lista.

factorial :: Int → Int

Factorial 0 = 1

Factorial n = n * factorial(n-1)

filterEven :: [Int] → [Int]

filterEven xs = filter(x → even x) xs

Ejemplo 3: Uso de Monadas para manejar E/S.

main :: IO()

main = do

PutStrLn "¿Cuál es tu nombre?"

name ← getLine

putStrLn ("Hola," ++ name ++ "!")