Universidad del Valle

Proyecto Final – Fundamentos de Lenguajes de Programación
Lenguaje: FlowLang

Robinson Duque, Ph.D
robinson.duque@correounivalle.edu.co

Octubre de 2025

1. Introducción

El presente proyecto busca que los estudiantes implementen un lenguaje de programación interpretado denominado **FlowLang**, utilizando la librería **EOPL** de Racket. Este lenguaje está inspirado en los principios de lenguajes como JavaScript y Python, mantiene un modelo **no tipado** (las variables pueden cambiar de tipo dinámicamente) y una filosofía basada en **prototipos** en lugar de clases tradicionales. El presente proyecto tiene por objeto enfrentar a los estudiantes del curso:

- a la comprensión de la relación entre sintaxis, semántica y ambiente de evaluación
- a la implementación de un lenguaje de programación
- al análisis de estructuras sintácticas, de datos y de control de un lenguaje de programación para la implementación de un interpretador

FlowLang fue concebido para enseñar la fluidez del pensamiento algorítmico sin las barreras del tipado o la compilación estricta. Su lema es: *"Deja que el código fluya."*

Importante: Tenga en cuenta que las descripciones presentadas en este documento son bastante generales y solo pretenden dar un contexto que le permita a cada estudiante entender el proyecto y las características del lenguaje que debe desarrollar. Así mismo, se proponen algunas construcciones sintácticas, **es posible que se requiera modificar o adaptar la sintaxis para cumplir con los requerimientos.**

- El desarrollo del proyecto y la participación de los integrantes debe ser rastreable y verificable en todo momento (no podrán aparecer entregas de la nada o cambios completos del interpretador sin evidenciar un desarrollo contínuo). Para esto, cada grupo deberá crear un **repositorio privado** en **Github** y en caso que se le solicite deberá invitar al usuario **robinsonduque**. Recuerde incluir la información de los integrantes en el archivo Readme al igual que información relevante del proyecto.

  La gramática **debe ser definida en el repositorio** durante las primeras tres semanas después de la asignación de este proyecto.

- La gramática deberá contener ejemplos de cada producción utilizando llamados a `scan&parse`. Es decir, deberá contener ejemplos de cómo se crean las variables, procedimientos, invocación a procedimientos, etc.

2. FlowLang

FlowLang es un lenguaje de tipado dinámico, interpretado y de propósito general. Admite expresiones, estructuras de control, prototipos, listas, diccionarios, funciones y operaciones aritméticas. Los programas en FlowLang se ejecutan evaluando expresiones dentro de un entorno mutable.

2.1. Valores

- **Valores denotados:** Ref(valores expresados)
- **Valores expresados:** enteros, flotantes, complejos, cadenas de caracteres, booleanos (true, false), nulos, funciones, listas, diccionarios y prototipos.

Sugerencia: trabaje los valores enteros, flotantes desde la especificación léxica. Implemente las cadenas de caracteres, booleanos (true, false), nulos, funciones, listas, diccionarios y prototipos desde la especificación gramatical.

Los números, cadenas y booleanos son inmutables. Las listas, diccionarios y prototipos son mutables (pueden modificarse en tiempo de ejecución).

2.2. Sintaxis Gramatical (70 pts)

En FlowLang, casi todas las estructuras sintáticas son expresiones, y todas las expresiones producen un valor. Se propone para este proyecto que usted implemente un lenguaje de programación para lo cual se brindan algunas ideas de sintáxis básica pero usted es libre de proponer ajustes siempre y cuando cumpla con

las funcionalidades especificadas. La semántica del lenguaje estará determinada por las especificaciones en este proyecto.

```
<exp> ::= <identificador>
       |  <numero>
       |  <cadena>
       |  <bool>
       |  null # representación de valor faltante
       |  var <identificador> = <exp>
       |  const <identificador> = <exp>
       |  <identificador> = <exp>      # actualización
       |  func ({<identificador>}*) {
              {<exp>}*
              return <exp> }
       |  <identificador>(<args>)     # invocacion
       |  begin {<exp>}+(;) end  # secuenciación
       |  if <exp> then <exp>  else {<exp>} end
       |  switch ....
       |  while <exp> do <exp> done
       |  for <id> in <exp> do <exp> done
       |  [ {<exp>} *(,) ]     # listas
       |  { {<identificador>:<exp>} +(,) }#diccionarios
```

- **Identificador:** Son secuencias de caracteres alfanuméricos que comienzan con una letra.

```
<exp>  := <identificador>
<identificador>::=<letter>| {<letter>|0,...,9| }*
<letter> ::= A..Z | a..z
```

- **Variables y asignación:** No existen tipos declarados ni palabras clave como int o string. El valor asignado define el comportamiento de la variable. con una letra.

```
var x = 42;
var nombre = "Robinson";
var promedio = 4.5;
var activo = true;
```

Las variables (una vez declaradas con la palabra "var") pueden cambiar de tipo durante la ejecución:

```
var x = 42;
x = "Ahora soy un texto";
```

Observe que en este lenguaje, a diferencia del lenguaje del curso, no se utiliza una estructura *let* o *letrec* para la creación de variables y su uso posterior. Las variables, una vez creadas, podrán ser utilizadas de manera inmediata.

- **Definiciones:** Este lenguaje deberá permitir crear cualquier cantidad de variables (para esto deberá ajustar la gramática)

```
var x1 = a1, x2 = a2, x3 = a3;
const y1 = b1, y2 = b2;
```

Una definición *var* introduce una colección de variables actualizables y sus valores iniciales. Una definición *const* introduce una colección de constantes no actualizables y sus valores iniciales.**El intento de modificar una variable inmutable (constante), deberá generar un error.**

Nota: en FlowLang, las declaraciones múltiples se separan por comas, y cada asignación produce un valor.

- **Datos:** Definen las constantes del lenguaje y permiten crear nuevos datos y objetos.

```
<exp> ::= <numero>
      ::= <cadena>
      ::= <bool>

Ejemplos :
0, 1, -1, 9.5   numeros
" abc "         cadena
true, false     bool
```

FlowLang evalúa las condiciones según valor de verdad dinámico: **false, 0, “”, null** (se consideran falsos). Cualquier otro valor verdadero

- **Constructores de Datos Predefinidos:**

El abordaje de estructuras como listas, diccionarios y valores booleanos puede llevarse a cabo de diversas maneras (como en la primera parte de esta sección o como se muestra a continuación o de forma similar):

```
<exp>
::= <lista>
::= <diccionario>
::= <exp-bool>

<lista> ::= [{<exp>} *(,)]

<diccionario> ::= { {<identificador> :
<exp> }+(,) }

<exp-bool>
  ::= <pred-prim>(<exp> , <exp>)
  ::= <oper-bin-bool>(<exp-bool>, <exp-bool>)
  ::= <bool>
  ::= <oper-un-bool>(<exp-bool>)

<pred-prim> ::= < | > | <= | >= | == | <>
<oper-bin-bool> ::= and|or
<oper-un-bool> ::= not
```

- **Primitivas para booleanos:** todas las operaciones que se muestra en la gramática anterior.

```
var a = 10;
var b = 5;

if (a > b and not (b == 0)) {
    print("a es mayor y b no es cero");
}
```

Las primitivas podrán implementarse con notación infija o prefija.

- **Primitivas aritméticas para enteros, flotantes:** $+, -, *, \%, /, add1, sub1$. Las primitivas podrán implementarse con notación infija o prefija.

- **Primitivas aritméticas para números complejoss:** $+, -, *, /$. Las primitivas podrán implementarse con notación infija o prefija.
- **Primitivas sobre cadenas:** longitud, concatenar. Las primitivas podrán implementarse con notación infija o prefija.
- **Primitivas sobre listas:** las listas en este lenguaje son una estructura de datos mutable, es decir, sus valores pueden ser actualizados en algún momento durante la ejecución de un programa. Las primitivas podrán implementarse con notación infija o prefija. Se deben crear primitivas que simulen el comportamiento de: `vacio?`, `vacio`, `crear-lista`, `lista?`, `cabeza`, `cola`, `append`, `ref-list`, `set-list`.
- **Primitivas sobre diccionarios:** Los diccionarios son estructuras mutables compuestas por conjuntos de claves y valores. Se debe extender el lenguaje y agregar manejo de registros. Se deben crear primitivas que simulen el comportamiento de: `diccionario?`, `crear-diccionario`, `ref-diccionario`, `set-diccionario`, `claves`, `valores`.
- **Estructuras de control:** el lenguaje deberá permitir el uso de estructuras de control para condicionales y switch (como el del primer parcial).

```
const edad = 15;

if edad > 18 {
    print("Mayor de edad");
} else {
    print("Menor de edad");
}
```

  Se debe agregar funcionalidad al lenguaje para que permita "imprimir" resultados por salida estándar tipo `print`.
- **Iteración:** el lenguaje debe permitir la defición de estructuras de repetición tipo `while y for` sobre estructuras iterables (como el for de python). Por ejemplo:

```
var mi_lista = [5,6,2,3];
var valor = 1.0;

for i in mi_lista{
    print(i)
}

while valor < 4.0 {
    valor = valor + 0.1;
}
```

- **Definición/invocación de funciones:** el lenguaje debe permitir la creación/invocación de funciones que retornan un valor al ser invocados. El paso de parámetros será por valor (para valores numéricos, cadenas, procedimientos) y por referencia (para listas y diccionarios) similar a como sucede en Python.

  Las funciones se declaran con func y pueden devolver cualquier tipo o incluso nada.Si una función no tiene return, devuelve automáticamente null. Las funciones pueden recibir cualquier tipo de argumento, incluso listas u objetos.

```
func sumar(a, b) {
    return a + b;
}

func saludar(nombre) {
    print("Hola, " + nombre); # retorna null
}
```

- **Definición/invocación de funciones recursivas:** el lenguaje debe permitir la creación/invocación de procedimientos que pueden invocarse recursivamente. El paso de parámetros será por valor (para valores numéricos, caracteres, cadenas, procedimientos, tuplas) y por referencia (para listas y registros) similar a como sucede en Python.

```
func factorial(n) {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
print(factorial(5)); # 120
```

- **Secuenciación:** el lenguaje deberá permitir expresiones para la creación de bloques de instrucciones ya sea por defecto o a través del uso de alguna expresión como el "begin".

2.3. Prototipos (30 pts)

FlowLang utiliza un modelo de objetos basado en **prototipos**, similar a JavaScript. No existen clases, sino estructuras que heredan propiedades directamente de otros objetos. El mecanismo principal es la función `clone(obj)`, que crea un nuevo objeto enlazado al prototipo `obj`. De este modo, cualquier cambio en el prototipo se refleja en sus clones, a menos que la propiedad sea sobrescrita localmente.

```
prototipo persona = {
    nombre: "Ana",
    edad: 25,
    saludar: func() {
        print("Hola, soy " + this.nombre)
    }
}

persona.saludar() # Hola, soy Ana
```

Ejemplo de herencia por prototipos:

```
prototipo estudiante = clone(persona);
estudiante.promedio = 4.3;
```

```
estudiante.saludar = func() {
    print("Soy " + this.nombre + " y tengo promedio " +
      this.promedio)
}

estudiante.saludar() #Soy Ana y tengo promedio 4.3
```

```
prototipo vehiculo = {
    encender: func() { print("Vehículo encendido"); }
}

prototipo carro = clone(vehiculo);
carro.modelo = "Sedán";
carro.encender = func() {
    print("Encendiendo carro modelo " + this.modelo);
}

vehiculo.encender(); # Vehículo encendido
carro.encender(); # Encendiendo carro modelo Sedán
```

La función clone(obj) crea una nueva referencia con el mismo prototipo. this siempre apunta al objeto que recibe la llamada.

Ejemplo con prototipos anidados (herencia en cadena):

```
prototipo vehiculo = {
    encender: func() { print("Vehículo encendido"); },
    mover: func() { print("El vehículo se mueve"); }
}

prototipo carro = clone(vehiculo);
carro.modelo = "Sedán";
carro.encender = func() {
    print("Encendiendo carro modelo " + this.modelo);
}

prototipo electrico = clone(carro);
electrico.bateria = 100;
electrico.recargar = func() {
    this.bateria = 100;
    print("Batería recargada completamente");
}

electrico.encender(); #Encendiendo carro modelo Sedán
electrico.recargar(); #Batería cargada completamente
```

Ejemplo de comportamiento compartido:

```
prototipo sensor = {
    encendido: falso,
    activar: func() { this.encendido = verdadero; },
    estado: func() {
        if this.encendido {
            print("Sensor activo");
        } else {
            print("Sensor apagado");
        }
    }
}

prototipo sensor1 = clone(sensor);
prototipo sensor2 = clone(sensor);

sensor1.activar();
sensor1.estado(); ;; Sensor activo
sensor2.estado(); ;; Sensor apagado
```

3. Evaluación

El proyecto podrá ser realizado en los grupos ya definidos utilizando la librería SLLGEN de Dr Racket. Este debe ser sustentado y cada persona del grupo obtendrá una nota entre 0 y 1 (por sustentación), la cual se multiplicará por la nota obtenida en el proyecto (la sustentación se llevará a cabo en el campus virtual).

El interpretador podrá contener algunas variaciones simples del lenguaje base. Dado el caso que un grupo se presente con un lenguaje distinto a la gramática definida para el interpretador (o generado completamente con IA o sin justificación del uso de estructuras/optimizaciones avanzadas en interpretadores o compiladores), obtendrán una nota de 0 en la sustentación asociada con dicho interpretador.

4. Adendo - Listas

Las listas en **FlowLang** son estructuras dinámicas que almacenan secuencias ordenadas de elementos. Las siguientes primitivas permiten crear, acceder y modificar listas. Su comportamiento se inspira en `Racket` y `Python`, pero se mantiene una sintaxis más legible.

4.0.1. `vacio`

Representa la lista vacía. Es un valor constante que indica la ausencia de elementos.

```
var lista = vacio;
```

4.0.2. `vacio?(lst)`

Devuelve `true` si la lista está vacía, `false` en caso contrario.

```
var lista = vacio;
print(vacio?(lista));                 # true
print(vacio?(crear-lista(1, vacio)));  # false
```

4.0.3. `crear-lista(elem, lst)`

Crea una nueva lista añadiendo un elemento al inicio de `lst`. Equivale al comportamiento de `cons` en Racket.

```
var lista = crear-lista(3, vacio);
lista = crear-lista(2, lista);
lista = crear-lista(1, lista);
print(lista);   # [1, 2, 3]
```

4.0.4. `lista?(x)`

Devuelve `true` si el valor `x` es una lista válida.

```
var a = crear-lista(5, vacio);
var b = 42;

print(lista?(a));   # true
print(lista?(b));   # false
```

4.0.5. `cabeza(lst)`

Devuelve el primer elemento de la lista. Si está vacía, devuelve `nulo` o genera un error.

```
var lista = crear-lista("A", crear-lista("B", vacio));
print(cabeza(lista));   # "A"
```

4.0.6. `cola(lst)`

Devuelve una nueva lista con todos los elementos excepto el primero. Si hay un solo elemento, devuelve `vacio`.

```
var lista = crear-lista(1, crear-lista(2, crear-lista
    (3, vacio)));
print(cola(lista));   # [2, 3]
```

4.0.7. `append(lst1, lst2)`

Concatena las listas `lst1` y `lst2`, devolviendo una nueva lista.

```
var a = crear-lista(1, crear-lista(2, vacio));
var b = crear-lista(3, crear-lista(4, vacio));
var c = append(a, b);
print(c);   # [1, 2, 3, 4]
```

4.0.8. `ref-list(lst, i)`

Devuelve el elemento en la posición `i` (índices desde 0). Si el índice es inválido, devuelve `nulo`.

```
var l = crear-lista("a", crear-lista("b", crear-lista("
    c", vacio)));
print(ref-list(l, 1));   # "b"
```

4.0.9. `set-list(lst, i, valor)`

Reemplaza el elemento en la posición `i` por `valor`. Devuelve la lista modificada.

```
var l = crear-lista(1, crear-lista(2, crear-lista(3,
    vacio)));
l = set-list(l, 1, 99);
print(l);   # [1, 99, 3]
```

4.0.10. Nota semántica

Internamente, la lista puede representarse como una estructura enlazada:

$$\textit{crear-lista}(x, xs) \equiv (x.xs)$$

Por ejemplo:

```
crear-lista(1, crear-lista(2, crear-lista(3, vacio)))
```

se interpreta como:

$$(1.(2.(3.\textit{vacio})))$$

Esto permite que las funciones `cabeza` y `cola` se implementen de forma recursiva.

5. Adendo - Diccionarios

Los **diccionarios** en **FlowLang** son estructuras que almacenan pares `(clave, valor)`. Permiten acceder rápidamente a los valores asociados a una clave y son útiles para representar datos estructurados. Estas primitivas simulan el comportamiento típico de estructuras tipo mapa o hash en otros lenguajes.

5.0.1. `crear-diccionario()`

Crea un nuevo diccionario vacío. Equivale a la construcción `{}` en lenguajes como Python o JavaScript.

```
var d = crear-diccionario();
print(d);   # {}
```

También puede inicializarse con pares clave-valor opcionales:

```
var d = crear-diccionario("nombre", "Ana", "edad", 34);
print(d);   # {"nombre": "Ana", "edad": 34}
```

5.0.2. `diccionario?(x)`

Devuelve `true` si el valor `x` es un diccionario válido, `false` en caso contrario.

```
var d = crear-diccionario();
var x = 42;

print(diccionario?(d));   # true
print(diccionario?(x));   # false
```

5.0.3. `ref-diccionario(dic, clave)`

Devuelve el valor asociado a la `clave` dentro del diccionario `dic`. Si la clave no existe, devuelve `nulo`.

```
var d = crear-diccionario("nombre", "Ana", "edad", 34);
print(ref-diccionario(d, "nombre"));   # "Ana"
print(ref-diccionario(d, "altura"));   # nulo
```

5.0.4. `set-diccionario(dic, clave, valor)`

Asigna un nuevo valor a la `clave` en el diccionario `dic`. Si la clave no existe, se crea automáticamente. Devuelve el diccionario actualizado.

```
var d = crear-diccionario("nombre", "Ana");
d = set-diccionario(d, "edad", 34);
d = set-diccionario(d, "nombre", "Ana María");

print(d);   # {"nombre": "Ana María", "edad": 34}
```

5.0.5. `claves`

Devuelve una lista con todas las claves de un diccionario. El orden no está garantizado, ya que depende de la implementación interna.

```
var pacientes = crear-diccionario(
  "id", 101,
  "nombre", "Carlos",
  "diagnostico", "Hipertension"
)
claves(pacientes)
# Resultado: ["id", "nombre", "diagnostico"]
```

5.0.6. `valores`

Devuelve una lista con todos los valores almacenados en un diccionario, en el mismo orden en que aparecen las claves.

```
var pacientes = crear-diccionario(
  "id", 101,
  "nombre", "Carlos",
  "diagnostico", "Hipertension"
)
valores(pacientes)
# Resultado: [101, "Carlos", "Hipertension"]
```

5.0.7. Nota semántica

Un diccionario puede representarse internamente como una colección de pares `(clave, valor)`:

$$dic = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$$

La operación `ref-diccionario(dic, clave)` busca secuencialmente la clave:

$$ref\text{-}diccionario(dic, k) = \begin{cases} v_i & \text{si } (k_i = k) \\ nulo & \text{si no se encuentra} \end{cases}$$

Mientras que `set-diccionario(dic, clave, valor)` reemplaza o inserta el par correspondiente.

6. Adendo - Primitivas de Control y Recursión

Las **estructuras de control** en **FlowLang** permiten modificar el flujo de ejecución de los programas, mientras que las **funciones recursivas** posibilitan expresar cálculos repetitivos de manera declarativa.

Estas primitivas deben implementarse como expresiones evaluables que retornan un valor o modifican el entorno según corresponda.

6.0.1. `if ... then ... else ... end`

Evalúa una condición y ejecuta uno de los dos bloques según su resultado lógico. La condición se considera verdadera si su valor no es `false`, `0`, o `null`.

```
var edad = 20;

if edad >= 18 then
    print("Mayor de edad");
else
    print("Menor de edad");
end
```

También puede anidarse:

```
if edad < 13 then
    print("Niño");
else if edad < 18 then
    print("Adolescente");
else
    print("Adulto");
end
```

6.0.2. `switch`

Evalúa una expresión y selecciona entre múltiples casos según su valor. Debe incluir al menos una cláusula `default` para manejar el caso por omisión.

```
var color = "rojo";

switch color {
    case "rojo": print("Detente");
    case "amarillo": print("Precaución");
    case "verde": print("Sigue");
    default: print("Color desconocido");
}
```

6.0.3. `while ... do ... done`

Repite un bloque de código mientras la condición sea verdadera. Ideal para ciclos donde el número de iteraciones no se conoce previamente.

```
var contador = 0;

while contador < 5 do
    print("Iteración " + contador);
    contador = contador + 1;
done
```

6.0.4. `for ... in ... do ... done`

Itera sobre los elementos de una lista o estructura iterable. El iterador toma el valor de cada elemento en cada paso.

```
var numeros = [1; 2; 3; 4; 5];

for n in numeros do
    print("Elemento: " + n);
done
```

Si la lista está vacía, el bloque simplemente no se ejecuta.

6.0.5. `func ... { ... return ... }`

Define una función (o procedimiento) con cero o más parámetros. Las funciones pueden anidarse y retornar cualquier tipo de valor, incluyendo listas o diccionarios. Si no se incluye `return`, la función devuelve automáticamente `null`.

```
func cuadrado(x) {
    return x * x;
}

print(cuadrado(4));   # 16
```

6.0.6. `funciones recursivas`

Las funciones pueden llamarse a sí mismas para resolver problemas que se definen de manera repetitiva. El paso de parámetros es por valor (para datos simples) y por referencia (para estructuras mutables como listas o diccionarios).

```
# Ejemplo: factorial recursivo
func factorial(n) {
    if n <= 1 then
        return 1;
    else
        return n * factorial(n - 1);
    end
}

print(factorial(5));   # 120
```

Otro ejemplo clásico es el cálculo de números de Fibonacci:

```
func fib(n) {
    if n <= 1 then
        return n;
    else
        return fib(n-1) + fib(n-2);
    end
}

print(fib(6));   # 8
```

6.0.7. `begin ... end`

Esta estructura es opcional ya que depende de cómo se defina la gramática. Permite agrupar múltiples

expresiones en un bloque secuencial, evaluadas en orden. El valor del bloque es el resultado de la última expresión.

```
begin
    var x = 5;
    var y = 10;
    var z = x + y;
    print("Suma: " + z);
end
```

6.0.8. Nota semántica

Cada estructura de control en **FlowLang** debe evaluarse como una expresión. Por ejemplo:

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

se evalúa como:

$$\text{eval}(e_1, env) = \begin{cases} \text{eval}(e_2, env) & \text{si } e_1 \text{ es verdadero} \\ \text{eval}(e_3, env) & \text{si } e_1 \text{ es falso} \end{cases}$$

Esto implica que incluso las estructuras condicionales o iterativas pueden utilizarse dentro de otras expresiones, conservando la naturaleza expresiva y funcional del lenguaje.