Universidad San Carlos de Guatemala
Facultad de ingeniería.
Ingeniería en ciencias y sistemas

Título del Proyecto:

JavaLang Interpreter

PONDERACIÓN: 40

Horas Aproximadas: 35

Índice

1. Resumen Ejecutivo

El presente proyecto consiste en el desarrollo de un intérprete funcional para el lenguaje JavaLang, un lenguaje académico con una sintaxis inspirada en Java, diseñado para fines educativos en el estudio de compiladores.

El objetivo es construir un sistema capaz de analizar, validar e interpretar programas escritos en JavaLang, permitiendo a los estudiantes comprender e implementar los principales componentes de un compilador. El intérprete será desarrollado en C utilizando Flex y Bison, e incluirá una interfaz gráfica que facilite la escritura, análisis y ejecución del código fuente.

El proyecto busca enfrentar la dificultad de visualizar y aplicar de manera práctica los conceptos teóricos relacionados con compiladores como el análisis léxico, sintáctico y semántico—, que suelen ser abstractos y complejos para los estudiantes.

Se propone el desarrollo de una herramienta completa que integre:

- Un analizador léxico y sintáctico generados con Flex y Bison.
- Un análisis semántico para asegurar la coherencia del código.
- La generación y recorrido del AST para la ejecución de programas.
- Una interfaz gráfica con funcionalidades de edición, ejecución y visualización de reportes.

El sistema aceptará archivos con extensión .usl y generará reportes detallados del AST, tabla de símbolos y errores detectados, brindando una experiencia interactiva y didáctica para el aprendizaje de los fundamentos de los compiladores.

2. Competencia que desarrollaremos

- El estudiante comprende las fases de un compilador a través de diagramas de flujo y modelos teóricos con ejemplos de compilación real en lenguajes como C.

- El estudiante utiliza herramientas generadoras de lexers y parsers tales como Flex, Bison en proyectos de análisis léxico y sintáctico

- El estudiante explica la eliminación de recursión por la izquierda con ejemplos paso a paso para preparar gramáticas para análisis sintáctico

3. Objetivos del Aprendizaje

3.1 Objetivo General

El estudiante será capaz de diseñar e implementar un intérprete para un lenguaje con sintaxis similar a Java, utilizando técnicas de análisis léxico, sintáctico y semántico, además de construir un AST y generar reportes asociados al código fuente procesado.

3.2 Objetivos Específicos

Al finalizar el proyecto, los estudiantes deberán ser capaces de:

1. Desarrollar una interfaz gráfica funcional multiplataforma que permita al usuario crear, editar y ejecutar archivos .usl, visualizar los resultados en consola y acceder a reportes generados durante el proceso de análisis y ejecución.

2. Diseñar y construir la gramática del lenguaje JavaLang utilizando Bison, permitiendo la generación automática de analizadores léxicos y sintácticos conforme a las reglas definidas por la sintaxis del lenguaje.

3. Implementar un sistema de análisis semántico y recorrido del AST, capaz de validar estructuras, tipos de datos y contextos del código fuente, así como de generar reportes de errores, tabla de símbolos y árbol de sintaxis abstracta.

4. Enunciado del Proyecto

4.1 Descripción del problema a resolver

En el estudio práctico de compiladores, los conceptos de análisis léxico, sintáctico y semántico suelen permanecer en el ámbito teórico hasta la implementación de un proyecto completo. Esto dificulta a los estudiantes visualizar cómo cada fase interactúa y cómo un lenguaje real procesa construcciones comunes de programación (variables, estructuras de control, arreglos, objetos, etc.).

El proyecto JavaLang resolverá esta brecha al proporcionar un intérprete que permita escribir y ejecutar código con sintaxis tipo Java, haciendo tangible el flujo completo desde el código fuente hasta la ejecución y generación de reportes.

4.2 Alcance del proyecto

Alcance obligatorio (Mínimos para entrega del proyecto)

Los estudiantes deberán implementar y poner a prueba en JavaLang las siguientes funcionalidades del lenguaje Java (en la hoja de calificación se marcan con **obligatorio**):

- Editor de código: La aplicación debe permitir crear, abrir, editar y guardar archivos con extensión .usl.
- Análisis léxico y sintáctico: generado con Flex/Bison para el lenguaje JavaLang.
- Análisis semántico: validación de tipos, declaración y alcance (scope) de identificadores, manejo de entornos.
- Ejecución: el intérprete localiza public static void main() y ejecuta de forma secuencial.
- Recorrido del AST: construcción y evaluación del Árbol de Sintaxis Abstracta.

Funcionalidades del lenguaje obligatorias:

- **Básicas**
  - Declaración de variables.
  - Asignación de variables.
  - Operaciones aritméticas.
  - Operaciones relacionales.
  - Operaciones lógicas.
  - Impresión: System.out.println()
  - Manejo de valor nulo (null).
  - Sintaxis: uso de ; para finalizar expresiones.
- **Intermedias**
  - Manejo de entornos (alcance/ámbitos).

    - Condicionales: if, if-else, else.
    - Bucles: while, for clásico, forEach.
    - Selección: switch/case.
    - Control de flujo: break, continue.
- **Funciones**
    - Funciones no recursivas sin parámetros.
    - Funciones no recursivas con parámetros.
- **Arreglos**
    - Creación de arreglos.
    - Acceso de elementos por índice.
    - Función add.
    - Tamaño de arreglos (length).
- **Reportes**
    - Tabla de símbolos.
    - Reporte de errores.
    - Reporte del AST.

Validación de funcionalidades obligatorias

Una funcionalidad obligatoria se considera terminada si, al ejecutar el intérprete con su archivo de entrada de aceptación, se cumplen todas estas condiciones:

- Ejecución: El programa finaliza sin errores léxicos, sintácticos y semánticos
- Salida: La salida de la ejecución coincide exactamente con lo previsto/definido para esa funcionalidad.

Si una o más entradas obligatorias fallan, se considera como no terminado la correspondiente funcionalidad.

Proceso de evaluación

El proceso se realizará en dos pasos:

Prueba de comprobación (archivo sencillo): se entregará un archivo de entrada simple cuyo objetivo es verificar que las funcionalidades obligatorias están implementadas de forma funcional. Si el programa no supera esta prueba, las funcionalidades correspondientes se consideran no terminadas y no se avanza al siguiente paso.

Prueba de calificación (archivo complejo): si la comprobación anterior es satisfactoria, se proporcionará un archivo de prueba más complejo que determinará la puntuación de cada funcionalidad. La nota refleja el comportamiento real sobre esa prueba.

4.3 Requerimientos tecnicos

Los estudiantes deberán emplear las siguientes tecnologías y herramientas en el desarrollo de JavaLang:

- Lenguaje de entrada
    - Java

- Lenguaje de implementación
    - C.
- Herramientas de construcción y dependencias
    - Makefile (o CMake) para compilación y enlace.
- Generación de analizadores
    - Flex para el análisis léxico.
    - Bison para el análisis sintáctico.
- Interfaz gráfica de usuario
    - Gtk3 (recomendado), queda libre el uso de otra biblioteca.
    - Contendrá:
        - Editor de texto con resaltado básico de sintaxis.
        - Panel de consola para salida y errores.
        - Paneles para visualizar: AST (como texto estructurado o diagrama), tabla de símbolos y lista de errores.

4.4 Generalidades del lenguaje JavaLang

4.4.1 Identificadores

Un identificador es una combinación de letras, dígitos o guión bajo, que debe iniciar con letra o **_**; no puede comenzar con número ni contener caracteres especiales como **. $, -**

4.4.2 Case sensitive

El lenguaje distingue mayúsculas de minúsculas en identificadores y palabras reservadas (por ejemplo, **if ≠ IF**)

4.4.3 Comentarios

```
// Esto es un comentario de una línea

/*

    Esto es un comentario multilínea

*/
```

Tipos estáticos

El lenguaje JavaLang no soportará múltiples asignaciones de diferentes tipos para una variable, por lo tanto si una variable es asignada con un tipo, solo será posible asignar valores de ese tipo a lo largo de la ejecución, sí alguna variable se le asignase un valor que no corresponde a su tipo declarado, el programa debe mostrar un mensaje detallando el error.

Tipos de datos primitivos

Se utilizan los siguientes tipos de datos primitivos.

Tipo	Descripción	Valor por defecto
int	Enteros de 32 bits, valor mínimo -2^31 y valor máximo 2^31 - 1	0
float	Punto flotante (32 bits), su notación es explícita con la letra f al final. Rango superior 3.4028235E38. Rango inferior 1.4E-45.	0.0f
boolean	true/false	FALSE
char	Carácter Unicode, el valor debe ir en comillas simples.	'\u0000'
String	Cadena de texto, los valores de la cadena están entre comillas dobles.	null
double	Punto flotante de 64 bits no teniendo notación explícita, es decir sin letras al final. Rango superior 1.7976931348623157E308. Rango inferior 4.9E-324.	0.0

Tipos compuestos

Solo arreglos unidimensionales o multidimensionales; no se permiten clases.

Valor nulo (null)

En el lenguaje Javalang se utiliza la palabra reservada **null** para indicar que una variable objeto no contiene valor, es decir fue declarada pero sin ningún valor. Por defecto cuando se declara un objeto así “MiObjeto miVariable;” siempre tendrá valor null y también es equivalente a declarar así “MiObjeto miVariable = null;”. Si se castea con una expresión que resulta en null el valor seguirá siendo null, ejemplo “String miVariable2 = (String) null;” al final será null.

Secuencias de escape

Secuencia	Definición
\"	Comilla Doble
\\	Barra invertida
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación

El comportamiento de “\n” y “\r” será un solo salto de línea en consola en caso no estén juntos, sí están juntos en el orden de “\r\n” seguirá realizando un salto de línea en lugar de dos. Ejemplo:

```
System.out.println(“Hola\n mundo”);
System.out.println(“Hola\r mundo”);
System.out.println(“Hola\r\n mundo”);
/* Salida:
    Hola
     mundo
    Hola
     mundo
    Hola
     mundo
*/
```

4.5 Sintaxis del lenguaje JavaLang

Bloques de sentencia

Será un conjunto de sentencias delimitado por llaves “{ }”, cuando se haga referencia a esto querrá decir que se está definiendo un ámbito local con todo y sus posibles instrucciones y que tiene acceso a las variables del ámbito global, además las variables declaradas en dicho ámbito únicamente podrán ser utilizadas en este ámbito o en posibles ámbitos anidados.

Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis “()”.

3- (1 + 3) * 32 / 90 // 1.5

Variables

Una variable es un elemento de datos cuyo valor puede cambiar durante el curso de la ejecución de un programa **siempre y cuando sea el mismo tipo de dato.** Una variable cuenta con un nombre y un valor, los nombres de variables no pueden coincidir con una palabra reservada.

Para poder utilizar una variable se tiene que definir previamente, la declaración nos permite crear una variable y asignarle un valor o sin valor.

Además la definición de tipo durante la declaración debe ser explícita y no puede cambiar.

Constantes

final <tipo> <identificador> = <expresión>;

- Deben inicializarse al declararse
- No pueden reasignarse; intentar hacerlo produce error

4.5.1 Operaciones

Las operaciones se especifican en esta sección y el resultado está determinado por el **casting**.

4.5.1.1 Casting

Consiste en la conversión de tipos de datos primitivos compatibles entre sí, para el caso del String se utiliza el parsing (ver sección correspondiente).

Widening Casting (Automáticamente) convierte un tipo pequeño a uno más grande.

char -> int -> float -> double

Ejemplo:

```
int counter = 0;
float counter2 = counter; // Automaticamente int -> float
```

Narrowing Casting (Manualmente) convierte un tipo grande a uno más pequeño.

double -> float -> int -> char

Ejemplo:

```
float counter = 0.5f;
int counter2 = (int) counter; //float -> int
System.out.println(counter2); //Salida 0, note redondea hacia abajo
```

4.5.1.2 Operaciones aritméticas

En base a la definición anterior se puede definir que cualquier operación del tipo suma (+), resta (-), multiplicación (*), división (/), Módulo (%), se puede realizar con los tipos: int, float, double, char.

Para la operación de la suma se tiene un caso especial con el tipo String pudiendo operar con otra cadena, quedando la unión de las dos cadenas como resultado. Ejemplo:

```
String cadena = String.valueOf(123) + " mundo";
System.out.println(cadena); //Salida: 123 mundo
```

Operadores de Bit a Bit

Los operadores de bits se utilizan para realizar operaciones a nivel de bits en valores enteros (int), en valores negativos se utiliza el complemento a dos y se opera de la misma forma.

Operador	Descripción	Ejemplo

&	Operador de bit AND, son multiplicaciones de bits, siendo 1 x 1 = 1 y 1 x 0 = 0.	int a = 5; // 0101
		int b = 3; // 0011
		System.out.println(a & b); // Salida: 1 (0001)

|	Operador de bit OR inclusivo, siendo resultado 1 en cualquier combinación que tenga un valor 1 de operando.	int a = 5; // 0101
		int b = 3; // 0011
		System.out.println(a | b); // Salida: 7 (0111)

^	Operador de bit XOR o OR exclusivo, si el valor de los operandos son iguales el resultado es 0.	int a = 5; // 0101
		int b = 3; // 0011
		System.out.println(a ^ b); // Salida: 6 (0110)

<<	Operador de desplazamiento a izquierdas, desplaza el valor binario a izquierdas tantas veces como indique el operando derecho añadiendo ceros.	int a = 3; // 0011
		int b = 1; // 0001
		System.out.println(a << b); // Salida: 6 (0110)
		// añade 0 a la derecha

>>	Operador de desplazamiento a derechas, desplaza el valor binario a derecha tantas veces indica el operando derecho y añade 0 si es positivo y 1 si es negativo.	int a = 3; // 0011
		int b = 1; // 0001
		System.out.println(a >> b); // Salida: 1 (0001)
		// añade 0 a la izquierda
		
		int a = -3; // 1101
		int b = 1; // 0001
		System.out.println(a >> b); //Salida: -2 (1110)
		// añade 1 a la izquierda

4.5.1.3 Operador de asignación

Se permiten los siguientes operadores de asignación para acotar las operaciones redundantes.

Operador	Ejemplo	Equivalente
=	x = 5	x = 5
+=	x += 3	x = x + 3

-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

4.5.1.4 Operaciones de comparación

Compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (**true**) o falsa (**false**). Los operandos pueden ser numéricos, Strings o lógicos, *permitiendo únicamente la comparación de expresiones del mismo tipo*.

Igualdad y desigualdad

Los operadores == y != para comparar valores primitivos. Para objetos, se usa .equals() para verificar la igualdad.

```
int a = 5;
int b = 5;
System.out.println(a == b); // true

String x = "hola";
String y = "hola";
System.out.println(x.equals(y)); // true
```

Relaciones

Los operadores relacionales comparan dos valores numéricos o caracteres:

- > mayor que
- < menor que
- >= mayor o igual que

- <= menor o igual que

```
int a = 10;
int b = 7;
System.out.println(a > b);  // true
System.out.println(a <= b); // false
```

4.5.1.5 Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones booleanas:

- && (AND): verdadero si ambas expresiones son verdaderas.
- || (OR): verdadero si al menos una es verdadera.
- ! (NOT): invierte el valor booleano.

```
boolean x = true;
boolean y = false;

System.out.println(x && y); // false
System.out.println(x || y); // true
System.out.println(!x);     // false
```

Sentencias de control de flujo

Las estructuras de control permiten regular el flujo de la ejecución del programa. Este flujo de ejecución se puede controlar mediante sentencias condicionales que realicen ramificaciones e iteraciones. Se debe considerar que estas sentencias se encontrarán únicamente dentro funciones.

If

```
int numero = 10;

if (numero > 5) {
    System.out.println("Mayor que cinco");
}
```

If else

```
if (numero % 2 == 0) {
    System.out.println("Par");
} else {
    System.out.println("Impar");
}
```

Else

La cláusula else se ejecuta si ninguna condición previa se cumple:

```
if (numero < 0) {
    System.out.println("Negativo");
} else {
    System.out.println("Cero o positivo");
}
```

Sentencia Switch - Case

```
int dia = 3;

switch (dia) {
    case 1:
        System.out.println("Lunes");
        break;
    case 2:
        System.out.println("Martes");
        break;
    case 3:
        System.out.println("Miércoles");
        break;
    default:
        System.out.println("Otro día");
}
```

Consideraciones:

- Cada case debe terminar con break para evitar la ejecución de los siguientes bloques.
- El default es opcional.

Sentencia While

El bloque while se ejecuta mientras la condición sea true.

```
int i = 0;

while (i < 5) {
    System.out.println(i);
    i++;
}
```

Sentencia For

```
for (int i = 0; i < 5; i++) {
    System.out.println("Valor: " + String.valueOf(i));
}


int[] numeros = {10, 20, 30};

for (int num : numeros) {
```

```
    System.out.println("Número: " + String.valueOf(num));
}
```

Sentencias de transferencia

- **Break**
  Esta sentencia termina el bucle actual ó sentencia switch y transfiere el control del programa a la siguiente sentencia a la sentencia de terminación de estos elementos.

  ```
  int counter = 0;
  while (true) { // Condición siempre verdadera
      System.out.println("Contador: " + String.valueOf(counter));
      counter++;
      break; // Rompe el bucle después de una iteración
  }
  ```

  Consideraciones:

  - Si se encuentra un break fuera de un ciclo y/o sentencia switch se considerará como un error.

  Continue
  Esta sentencia termina la ejecución de las sentencias de la iteración actual (en un bucle) y continúa la ejecución con la próxima iteración.

  ```
  int i = 0;
  int j = i;

  while (i < 2) {
      if (j == 0) {
          i = 1;
          i += 1;
          continue;
      }
      i += 1;
  }
  // i posee el valor de 2 al finalizar el ciclo
  ```

  Consideraciones:

  - Si se encuentra un continue fuera de un ciclo se considerará como un error.

  Return
  Sentencia que finaliza la ejecución de la función actual, puede o no especificar un valor para ser devuelto a quien llama a la función.

  ```
  public int funcion1() {
      return 1; // Retorna un valor entero
  }
  ```

```
public void funcion2() {
    return; // No retorna ningún valor
}
```

Estructuras de datos

- **Vectores**
  Arreglo unidimensional de elementos del mismo tipo.

  Sintaxis:
  <tipo>[] <identificador> = new <tipo>[<expresión>];

  <tipo>[] <identificador> = { elem1, elem2, … };

  Ejemplo:
  ```
  int[] nums = new int[5];
  int[] primes = { 2, 3, 5, 7, 11 };
  ```

  Consideraciones:

  - Índices van de 0 a length-1; acceder fuera de rango es error de índice.
  - Todos los elementos se inicializan al valor por defecto del tipo

- **Matrices**
  Arreglo bidimensional (vector de vectores)

  Sintaxis:
  <tipo>[][] <identificador> = new <tipo>[<filas>][<columnas>];

  <tipo>[][] <id> = {

  {valor00, valor01, …},

  {valor10, valor11, …},

  };

  Ejemplo:
  ```
  float[][] mat = new float[3][3];
  int[][] board = {
        { 0, 1, 0 },
        { 1, 0, 1 },
        { 0, 1, 0 }
  };
  ```

  Consideraciones:

  - Cada “fila” es a su vez un vector.

- Fuera de límite en cualquier dimensión genera error de índice

- **Acceso de elemento**
  Obtención de un valor concreto en un arreglo.

  Sintaxis:
  <id>[<índice>]; // vector

  <id>[<fila>][<col>]; // matriz

  Ejemplo:
  ```
  int x = prueba[2];
  float y = mat[1][2]; // segundo renglón, tercera columna
  ```

  Consideraciones:
  - El índice debe ser entero y respetar los límites del arreglo.
  - Cualquier acceso fuera de rango se considera error semántico

- **Array multidimensional**
  Arreglo multidimensional de elementos del mismo tipo.
  Sintaxis:
  <tipo> <nombre>[]…[] = new <tipo>[<dim1>][<dim2>]…[<dimN>];
  <tipo> <nombre>[]…[] = {
  { /* bloque 1 dimensión interna, anidado según dimensiones */ },
  { /* bloque 2 */ },
  …
  };
  - **[ ]…[ ]**: uno o más pares de corchetes tras el nombre, una por cada dimensión.
  - **new <tipo>[<dim1>][<dim2>]…[<dimN>]**: construcción con longitudes de cada dimensión (<dimX> es un literal entero).

  Ejemplo:
  ```
  int matriz2D[][] = new int[3][4];
  int cubo3D[][][] = new int[2][3][4];
  char tablaChars[][] = new char[10][10];
  int hipercubo4D[][][][] = new int[2][2][2][2];

  int matriz2D[][] = {
      { 1, 2, 3 },
      { 4, 5, 6 },
      { 7, 8, 9 }
  };
  ```

```
int cubo3D_2[][][] = {
    { // primer “bloque” de 3 filas de 4 columnas
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }
    },
    { // segundo “bloque” de 3 filas de 4 columnas
        { 13, 14, 15, 16 },
        { 17, 18, 19, 20 },
        { 21, 22, 23, 24 }
    }
};
```

Consideraciones:

- El arreglo multidimensional declarado con new se inicializa con todas las posiciones inicializadas al valor por defecto (0 para numéricos, '\u0000' para char).

- **Acceso array multidimensional**

  Permite leer o escribir en una posición específica de un arreglo con dos o más dimensiones.

  Sintaxis:

  <tipo> <nombre> = <array>[<índice1>][<índice2>]…[<índiceN>];

  Ejemplo:

```
int[][] matriz = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Leer la fila 1, columna 2 (valor 6)
int valor = matriz[1][2];

// Escribir en fila 0, columna 0
matriz[0][0] = 42;

int[][][] cubo = new int[2][3][4];
// Acceder a la posición [1][2][3]
int x = cubo[1][2][3];
```

Funciones
En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o Interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones y sentencias, que conforman el cuerpo de la función.

Se pueden pasar valores a una función y la función puede devolver un valor. Para devolver un valor específico, una función debe tener una sentencia return, que especifique el valor a devolver. Además la función debe tener especificado el tipo de valor que va a retornar.

- **Con parametros**

  Bloque de código nombrado que recibe uno o más argumentos.

  Sintaxis:
  <tipo> <nombre>(<tipo1> param1, <tipo2> param2, …) {

  <sentencias>

  }

  Ejemplo:
  ```
  int suma(int a, int b) {
    return a + b;
  }
  ```

  Consideraciones:

  - Cada parámetro incluye tipo y nombre, separados por comas.

- **Sin parametros**

  Bloque de código nombrado que no recibe argumentos.

  Sintaxis:

  <tipo> <nombre>() {

  <sentencias>

  }

  Ejemplo:

```
void saludo() {
   System.out.println("¡Hola!");
}
```

- **Funciones Recursivas**

Una función recursiva es aquella que se llama a sí misma para resolver un problema que puede descomponerse en subproblemas similares. Es importante definir una condición de parada para evitar ciclos infinitos.

Ejemplo:

```
public int factorial(int n) {
    if (n == 0 || n == 1) return 1;
    return n * factorial(n - 1);
}
```

- **Llamada de funciones**

Ejecución de una función previamente declarada.

Sintaxis:

<resultado> = <nombre>(arg1, arg2, …);

<nombre>(arg1, arg2, …); // si la función es void

Ejemplo:

```
int total = suma(3, 4);
saludo();
```

Funciones embebidas

System.out.println

Imprime la representación en texto de una expresión y añade un salto de línea.

Sintaxis:

System.out.println(<expresión>);

Ejemplo:

```
System.out.println("Resultado: " + total);
```

Consideraciones:

- Convierte implícitamente valores primitivos y objetos a String.

Parseo de Enteros

Java permite convertir una cadena de texto (String) en un valor numérico entero (int) usando métodos de la clase Integer.

Sintaxis:

int x = Integer.parseInt(<cadena>);

Ejemplo:

```
int edad = Integer.parseInt("25");
```

Consideraciones:

- Lanza error semántico si la cadena no es numérica.

Parseo de Flotantes

Para convertir cadenas a números de punto flotante (`float` o `double`), se utiliza la clase `Float` o `Double`.

Sintaxis:
double d = Double.parseDouble(<cadena>);

float f = Float.parseFloat(<cadena>);

Ejemplo:

```
double pi = Double.parseDouble("3.1416");
float temp = Float.parseFloat("36.5");
```

Consideraciones:

- Lanza error semántico si es un formato no valido

String.valueOf()

Genera un String que representa cualquier valor primitivo u objeto.

Sintaxis:
String s = String.valueOf(<expresión>);

Ejemplo:

```
String s1 = String.valueOf(123);
String s2 = String.valueOf(true);
```

Strings.join()

Une una serie de secuencias de caracteres (por ejemplo, literales o elementos de un arreglo) insertando entre ellas un delimitador que se especifique, y devuelve el resultado como un único String.

Sintaxis:

```
String resultado = String.join(delimitador, elemento1, elemento2, ..., elementoN);
// Arreglos
String[] resultado = String.join(delimitador, arregloDeStrings);
```

- **delimitador**: una cadena que se inserta entre cada par de elementos unidos.
- **elementoX**: cada CharSequence a unir (por ejemplo literales "a", "b"…).
- **arregloDeStrings**: un String[] cuyas posiciones se concatenarán en orden.

Ejemplos:

```
String s = String.join("-", "2025", "08", "03");
// s vale "2025-08-03"

String[] frutas = { "manzana", "banana", "cereza" };
String csv = String.join(",", frutas);
// csv vale "manzana,banana,cereza"

String nums = String.join(" | ", "10", "20", "30");
// nums vale "10 | 20 | 30"
```

- **Arrays.indexOf()**
  Bloque de código nombrado que recibe un arreglo y devuelve la posición de la primera ocurrencia (o -1 si no se encuentra).

  Sintaxis
  Arrays.indexOf(vector, clave);

  Ejemplo:

```
int[] numeros = { 2, 5, 7, 8, 11, 12 };
int clave = 7;
int indice = Arrays.indexOf(numeros, clave); // indice == 2
```

  Consideraciones:

  - El índice es 0-based (el primer elemento está en 0).
  - Si aparece varias veces, devuelve la primera ocurrencia.

- Retorna -1 si no se encuentra.

- **Array.length**

  Sintaxis:

  <arrayDeTipoPrimitivo>.length

  - <arrayDeTipoPrimitivo>: cualquier expresión que resulte en un arreglo (int[], double[], etc.)
  - .length: acceso a la propiedad que retorna un int con la cantidad total de elementos.

  Ejemplo:

```
int[] numeros = { 2, 5, 7, 8, 11, 12 };
int tam = numeros.length; // tam == 6
```

- **Array.add()**

  Añade un elemento al final de la lista

  Sintaxis:

  <arrayDeTipoPrimitivo>.add(<tipo elemento>)

  - <arrayDeTipoPrimitivo>: cualquier expresión cuyo tipo sea int[], double[], etc.
  - .add: operación que “inserta” el elemento al final.
  - <tipo elemento>: el valor primitivo que deseas incorporar.

  Ejemplo:

```
int[] numeros = { 2, 5, 7, 8 };
int nuevoVal = 11;

int[] ampliado = numeros.add(nuevoVal);
// ampliado == {2, 5, 7, 8, 11}
```

4.6 Reportes

JavaLang genera una serie de reportes sobre el proceso de análisis de los archivos de entrada. Los reportes son los siguientes:

4.6.1 Reporte de Errores

El Intérprete deberá ser capaz de detectar todos los errores que se encuentren durante el proceso de compilación. Todos los errores se deberán de recolectar y se mostrará un reporte de errores en el que, como mínimo, debe mostrarse el ámbito, su ubicación y una breve descripción de por qué se produjo.

No.	Descripción	Ámbito	Linea	Columna
1	No se puede dividir entre 0	Global	19	6
2	El símbolo “¬” no es aceptado en el lenguaje	Ackerman	55	2

4.6.2 Reporte de tabla de símbolos

Este reporte mostrará la tabla de símbolos después de la ejecución del archivo. Se deberán de mostrar todas las variables, funciones y procedimientos que fueron declarados, así como su tipo y toda la información que el estudiante considere necesaria para demostrar que el intérprete ejecutó correctamente el código de entrada.

ID	Tipo símbolo	Tipo dato	Ámbito	Linea	Columna
x	Variable	float	Global	6	2
Ackerman	Función	int	Global	15	1

4.6.3 Reporte de AST

En esta ocasión en este reporte se mostrará el árbol de análisis abstracto, que se produjo al analizar el archivo de entrada. Este debe de representarse como un grafo, se recomienda utilizar Graphviz para la implementación de dicho árbol. Sin embargo es libre de utilizar la herramienta que crea conveniente

4.7 Entregables

Tipo	Descripción
Prototipo funcional	Archivo ejecutable compilado en C del intérprete y la GUI de JavaLang, junto con instrucciones de lanzamiento. Permite crear, cargar, ejecutar y visualizar reportes.
Código fuente	Repositorio Git organizado (GitHub/GitLab) con todo el código C, especificaciones Flex/Bison y scripts de compilación..

Informe técnico	Documento (máx. 10 págs.) que describa: ● Gramática formal de JavaLang ● Diseño de módulos (lexer, parser, AST, semántico, UI) ● Decisiones de diseño y desafíos enfrentados

- **Prototipo funcional**
  - Archivo ejecutable de JavaLang con la GUI operativa.
  - Código fuente completo en el repositorio, incluyendo scripts de compilación y definiciones Flex/Bison.

- **Informe técnico**
  - Documento de máximo 10 páginas que describa:
    • Gramática formal de JavaLang y arquitectura general.
    • Implementación de módulos (lexer, parser, AST, semántico y GUI).
    • Retos técnicos encontrados y soluciones aplicadas.
    • Resultados de pruebas y métricas de cobertura.

- **Documentación de usuario**
  - Manual paso a paso para instalar y usar la herramienta con extensión .usl.
  - Instrucciones para crear, editar y ejecutar código, así como para interpretar los reportes (AST, tabla de símbolos y errores).
  - Capturas de pantalla de la interfaz y ejemplos de sesión de uso.

5. Metodología

Se detallan las fases y prácticas recomendadas para el desarrollo de JavaLang. Se sugiere adoptar un enfoque ágil organizando el trabajo en bloques de 1–2 semanas.

1. **Investigación y Preparación**
    - Revisar conceptos de teoría de compiladores: análisis léxico, sintáctico y semántico.
    - Estudiar ejemplos básicos de gramáticas en Flex/Bison y proyectos similares en C.
    - Configurar el entorno: Compilador C (gcc), Makefile, Flex, Bison, Editor de texto o IDE (Codeblocks) para C y repositorio Git.
2. **Diseño de la Gramática y Arquitectura**
    - Definir la gramática formal de JavaLang (.usl), cubriendo variables, estructuras de control, arreglos, funciones.
    - Esquematizar la arquitectura modular: paquetes lexer, parser, ast, semantic y ui.
    - Elaborar diagramas de clases y de flujo de procesamiento (AST, tabla de símbolos, errores).
3. **Desarrollo Iterativo**
    - **Bloque 1**:
        - Implementar y probar el analizador léxico con Flex.
    - **Bloque 2**:
        - Generar el parser con Bison y construir el AST.
    - **Bloque 3**:
        - Implementar el análisis semántico (tabla de símbolos, validaciones de tipos).
        - Crear casos de prueba para semántica y manejo de errores.
    - **Bloque 4**:

- Desarrollar la GUI en Gtk3 (o la biblioteca de su preferencia): editor, consola y paneles de reportes.

- Integrar el intérprete con la interfaz.

4. **Pruebas y Validación**

    - Realizar pruebas de usuario con ejemplos de código JavaLang para asegurar usabilidad de la GUI y claridad de reportes.

5. **Documentación y Entrega**

    - Preparar el informe técnico.

6. **Revisión y Retroalimentación**

    - Al final de cada bloque, realizar pruebas de lo elaborado.
    - Ajustar el plan de trabajo y priorizar correcciones identificadas antes de la entrega final.

6. Desarrollo de Habilidades Blandas

6.1 Proyectos Individuales

En los proyectos individuales, cada estudiante asume plenamente la planificación, ejecución y entrega, promoviendo no solo habilidades técnicas sino también Profesionales y de aprendizaje continuo.

6.1.1 Gestión del tiempo

Planificar tareas y establecer hitos propios para asegurar el avance constante y la entrega puntual de cada fase del intérprete.

6.1.2 Autonomía en la investigación

Buscar y seleccionar recursos, ejemplos de gramáticas y buenas prácticas para resolver dudas y guiar el diseño sin depender de supervisión constante.

6.1.3 Resolución creativa de problemas

Enfrentar errores en la gramática, el parser o la lógica de ejecución con iniciativas propias: diseñar casos de prueba, depurar paso a paso y ajustar la implementación.

6.1.4 Calidad de la documentación y pruebas

Mantener un README claro, escribir casos de prueba significativos y documentar cada módulo (lexer, parser, AST, semántico, GUI) para facilitar el mantenimiento y la evaluación.

7. Cronograma

El cronograma describe las etapas clave del proyecto, los plazos estimados para cada una, y el proceso de asignación, elaboración y calificación de las tareas. Los estudiantes deberán seguir este plan para asegurar que el proyecto avance de manera organizada y cumpla con los plazos establecidos. Cada fase incluye la asignación de tareas, el tiempo estimado para su elaboración, y el momento de su calificación.

Tipo	Fecha Inicio	Fecha Fin
Asignación de Proyecto	11/08/2025	15/08/2025
Elaboración	12/08/2025	12/09/2025
Calificación	13/09/2025	20/09/2025

8. Rúbrica de Calificación

8.1 Requisitos para optar a la calificación

Tema	**Descripción**	**Cumple (Sí/No)**
Cumplimiento de la tecnología establecida	El desarrollo debe haberse realizado en lenguaje C, usando Flex y Bison para el lexer/parser, Gtk3 para la GUI y sistema operativo Linux.	Si
Gestión y entregas del proyecto	- Todas los commits realizados disponibles en el repositorio GitHub.	Si
Documentación obligatoria	- Manual de usuario y plan de pruebas. - Diagrama de clases y de flujo de procesamiento (AST, tabla de símbolos).	Si
Pruebas y funcionalidad mínima	- El intérprete ejecuta correctamente los elementos obligatorios (**definido en la sección de alcance obligatorio**). - Evidencia de pruebas unitarias e integración.	Si
Entrega de prácticas	- La entrega de las 6 prácticas de laboratorio es requisito indispensable para tener derecho a calificación.	Si

Modificación de código	- Se indicará al estudiante que modifique parte del código; en caso de no lograrlo, no podrá ser calificado.	Sí

8.2 Resumen de Puntuaciones

Área	Puntos Totales	Puntos Obtenidos
1. Conocimiento		
Funcionalidad del Proyecto	90	
Sub-Total	**90**	
2. Reportes		
Reportes del proyecto	10	
Sub-Total	**100**	
TOTAL	**100**	

*La calificación debe incluir ambas áreas conocimientos y habilidades.

8.3 Detalle de la Calificación

Criterio	Descripción	Puntos Máximos	Puntuación Obtenida
1. Conocimiento			
1. Funcionalidad del Proyecto		**90**	
1.1 Funcionalidades Básicas (obligatorio)		**16**	
1.1.1 Declaración de variables		2	
1.1.2 Asignación de variables		2	
1.1.3 Operaciones Aritméticas		2	
1.1.4 Operaciones Relacionales		2	

1.1.5 Operaciones Lógicas		2	
1.1.6 System.out.println()		2	
1.1.7 Manejo de valor nulo		2	
1.1.7 Uso ; al finalizar las expresiones		2	
1.2 Funcionalidades Intermedias (obligatorio)		**23**	
1.2.1 Manejo de entornos		3	
1.2.2 If / If else / Else		3	
1.2.3 While		2	
1.2.4 For Clásico		3	
1.2.5 ForEach		3	
1.2.6 Switch/Case		3	
1.2.7 Break		3	
1.2.8 Continue		3	
1.3 Funciones		**26**	
1.3.1 Funciones no recursivas sin parámetros (obligatorio)		5	
1.3.2 Funciones no recursivas con parámetros (obligatorio)		5	
1.3.3 Funciones recursivas		10	
1.3.4 Parseo de enteros		2	
1.3.5 Parseo de flotantes		1.5	
1.3.6 String.valueOf()		0.5	
1.3.7 Strings.join()		0.5	
1.3.8 Arrays.indexOf()		0.5	
1.3.9 Array.length		0.5	
1.3.10 Array.add()		0.5	
1.4 Arreglos		**25**	

1.4.1 Creación de arreglos (obligatorio)		3	
1.4.2 Acceso de Elementos (obligatorio)		3	
1.4.3 Array Multidimensional		4	
1.4.4 Acceso Array Multidimensional		4	
1.4.5 Función para encontrar el índice de un elemento en un arreglo (Array.indexOf)		3	
1.4.6 Función Strings.join		2	
1.4.7 Tamaño de arreglos (length) (obligatorio)		3	
1.4.8 Función add (obligatorio)		3	
2.Reportes		10	
Tabla de símbolos (obligatorio)		2.5	
Reporte de errores (obligatorio)		2.5	
Reporte de AST (obligatorio)		5	
Sub-Total de Puntos		100	
Total		**100**	

8.4 Valores

En el desarrollo de JavaLang, se espera que cada estudiante demuestre los siguientes principios éticos y profesionales:

- **Originalidad del trabajo**
  Todo código, documentación y entregable debe ser producto del esfuerzo personal. Se valorará la creatividad y la aplicación directa de los conceptos del curso.

- **Prohibición de copia y plagio**
  Copiar total o parcialmente código, documentación o proyectos anteriores sin citar la fuente conlleva calificación de 0. Esto aplica tanto a trabajos de compañeros como a repositorios o ejemplos en Internet.

- **Uso responsable de recursos externos**
  Está permitido consultar bibliotecas, frameworks o ejemplos de código ajenos, siempre que se referencien correctamente en la documentación y se demuestre comprensión de su funcionamiento. Para dudas sobre la política de uso de terceros, consultar al catedrático.

- **Revisión y detección de similitudes**
  Se emplearán herramientas automatizadas y revisiones manuales para detectar código o documentación duplicada. Ante cualquier sospecha, el estudiante deberá explicar y demostrar el proceso de desarrollo. La imposibilidad de justificar el trabajo individual o en equipo resultará en calificación de 0.

Cualquier incumplimiento de estos valores será comunicado al catedrático, quien decidirá las sanciones académicas correspondientes.

8.5 Comentarios Generales

_____

_____

_____

_____