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

Título del Proyecto:

JavaLang Compiler

PONDERACIÓN: 60

Índice

1. Resumen Ejecutivo

El presente proyecto consiste en el desarrollo de un compilador 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 y traducir programas escritos en JavaLang a código ensamblador ARM64, permitiendo a los estudiantes comprender e implementar los principales componentes de un compilador.

El compilador será desarrollado en C utilizando Flex y Bison, e incluirá una interfaz gráfica que facilite la escritura, análisis y generación del código ensamblador.

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

El desarrollo de esta herramienta debe integrar lo siguiente:

- Un analizador léxico y sintáctico generados con Flex y Bison.
- Un análisis semántico para asegurar la coherencia del código.
- La generación de código en instrucciones del lenguaje ensamblador ARM64.
- Una interfaz gráfica como editor de código que contenga visualización de reportes.

El sistema aceptará archivos con extensión .usl y generará reportes del AST, tabla de símbolos y errores detectados.

2. Competencia que desarrollaremos

- El estudiante comprende el proceso completo de compilación desde código fuente hasta código ejecutable, incluyendo las representaciones intermedias.
- El estudiante utiliza herramientas generadoras de lexers y parsers tales como Flex y Bison en proyectos de análisis léxico y sintáctico.
- El estudiante genera código de bajo nivel y ensamblador a partir de plantillas de generación y asignación de registros considerando convenciones de llamada.

3. Objetivos del Aprendizaje

3.1 Objetivo General

El estudiante será capaz de diseñar e implementar un compilador completo que traduzca código JavaLang a código ensamblador ARM64.

3.2 Objetivos Específicos

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

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

2. Desarrollar un generador de código ARM64 que traduzca el código intermedio a instrucciones ensamblador válidas, considerando convenciones de llamada, manejo de registros y optimización básica.

4. Enunciado del Proyecto

4.1 Descripción del problema a resolver

En el estudio práctico de compiladores, muchos estudiantes logran comprender los conceptos teóricos de análisis léxico, sintáctico y semántico, pero rara vez experimentan el proceso completo que transforma un lenguaje de alto nivel en código ejecutable sobre una arquitectura real. Esta desconexión entre teoría y aplicación concreta limita la comprensión de cómo los lenguajes modernos son compilados y optimizados para plataformas específicas.

El proyecto JavaLang Compiler aborda esta problemática proporcionando una experiencia práctica integral: un sistema que no solo analiza y valida el código fuente de un lenguaje con sintaxis similar a Java, sino que además genera código ensamblador para la arquitectura ARM64. Esto permitirá a los estudiantes comprender de forma clara y tangible cómo se traduce una construcción de alto nivel en instrucciones de bajo nivel ejecutables, y cómo cada fase del compilador impacta el rendimiento y la corrección del programa resultante.

El enfoque del proyecto hace énfasis en cerrar la brecha entre los aspectos teóricos y su aplicación real, integrando una herramienta que permite visualizar el flujo completo de la compilación: desde la edición del código, su análisis y validación, hasta la generación del código ensamblador, todo acompañado de reportes detallados del proceso.

4.2 Requerimientos técnicos

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

- Lenguaje de implementación: C.
- Ejecución del compilador en cualquier distribución de Linux.
- Sistema de construcción (opcional): Makefile o CMake
- Generación de analizadores:
    - Flex para el análisis léxico.
    - Bison para el análisis sintáctico.
- Generación de código ensamblador
    - El compilador deberá generar código en lenguajes ensamblador ARM64, compatible con arquitecturas de 64 bits basadas en ARM (como Raspberry Pi o entornos simulados).
    - El código generado deberá seguir convenciones básicas de llamada y uso de registros de ARM64, incluyendo el manejo adecuado de la pila y los entornos en tiempo de ejecución.
    - El ensamblador generado deberá poder ser probado utilizando herramientas como:
        - GNU AArch64
        - Simuladores o emuladores que soporten ARM64.
- Interfaz gráfica de usuario

- Gtk3 (recomendado)
- Contendrá:
  - Editor de texto.
  - Crear, abrir y guardar archivos con extensión .usl.
  - Paneles para visualizar: AST (como texto estructurado o diagrama), tabla de símbolos y lista de errores.
  - Botón para compilar y generar los archivos de código ensamblador ARM64.

4.3 Generalidades del lenguaje JavaLang

4.3.1 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 **. $, -**

4.3.2 Case sensitive

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

4.3.3 Comentarios

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

4.3.4 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.

4.3.4.1 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	0.0f

	3.4028235E38. Rango inferior 1.4E-45.	
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

4.3.4.2 Tipos compuestos

Solo arreglos unidimensionales o multidimensionales; no se permiten clases.

4.3.4.3 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 una cadena así "String miVariable;" siempre tendrá valor null y también es equivalente a declarar así "String miVariable = null;".

4.3.4.4 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
*/
```

4.4 Sintaxis del lenguaje JavaLang

4.4.1 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.

4.4.2 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

4.4.3 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.

4.4.4 Constantes

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

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

4.4.5 Operaciones

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

4.4.5.1 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
```

4.4.5.2 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
```

4.4.5.3 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	int a = 5; // 0101

	exclusivo, si el valor de los operandos son iguales el resultado es 0.	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

4.4.5.4 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

4.4.5.5 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
```

4.4.5.6 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
```

4.4.6 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.

4.4.6.1 if

```
int numero = 10;
if (numero > 5) {
    System.out.println("Mayor que cinco");
}
```

4.4.6.2 if else

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

4.4.6.3 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");
}
```

4.4.6.4 else if

La cláusula else if se ejecuta si la cláusula anterior no se cumplió:

```
if (numero < 0) {
    System.out.println("Negativo");
} else if (numero > 0){
    System.out.println("Positivo");
}
```

4.4.6.5 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.

4.4.6.6 Sentencia While

El bloque while se ejecuta mientras la condición sea true.

```
int i = 0;

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

4.4.6.7 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));
}
```

4.4.7 Sentencias de transferencia

4.4.7.1 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.

4.4.7.2 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.

4.4.7.3 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.

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

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

4.4.8 Estructuras de datos

4.4.8.1 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

4.4.8.2 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

4.4.8.3 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

4.4.8.4 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).

4.4.8.5 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];
```

4.4.9 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.

4.4.9.1 Con parametros

Un bloque de código nombrado puede recibir uno o varios argumentos. Estos argumentos pueden pasarse de dos formas: por valor o por referencia.

- **Por valor:** se aplica únicamente a variables de tipo simple (int, boolean, char, double, float). En este caso, cualquier modificación realizada dentro de la función no afecta la variable original fuera de ella.

- **Por referencia:** se aplica a estructuras como *String* o arreglos. En este modo, los cambios realizados dentro de la función sí se reflejan en la variable original. Por ejemplo, al concatenar un *String* con otro, la variable pasada como argumento conservará la modificación.

Sintaxis:

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

   <sentencias>

}
```

Ejemplo:

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

Ejemplo por referencia:

```
void concatenar(String cadena) {
   cadena += " mundo";
}

public static void main() {
   String cadena = "Hola";
   concatenar(cadena);
   System.out.println(cadena) //salida: Hola mundo
}
```

Ejemplo por valor:

```
void sumar(int numero) {
    numero += 1;
}

public static void main() {
    int numero = 0;
    sumar(numero);
    System.out.println(numero) //salida: 0
}
```

Consideraciones:

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

4.4.9.2 Sin parametros

Bloque de código nombrado que no recibe argumentos.

Sintaxis:

```
<tipo> <nombre>() {

    <sentencias>

}
```

Ejemplo:

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

4.4.9.3 Funciones Recursivas

Una función recursiva es aquella que se llama a sí misma directa o indirectamente para resolver un problema que puede descomponerse en subproblemas más simples y de la misma naturaleza.

Ejemplo:

```
int factorial(int n) {

    if (n == 0 || n == 1) return 1;

    return n * factorial(n - 1);

}
```

4.4.9.4 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();
```

4.4.10 Funciones embebidas

4.4.10.1 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.

4.4.10.2 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.

4.4.10.3 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

4.4.10.4 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);
```

4.4.10.5 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"
```

4.4.10.6 Arrays.indexOf()

Bloque de código nombrado que recibe un arreglo y devuelve la posición de la primera ocurrencia (o -1 si no se encuentra). Se aplica a los arreglos de una dimensión.

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.

4.4.10.7 Array.length

Sintaxis:

<arrayDeTipoPrimitivo>.length

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

Ejemplo:

```
int[] numeros = { 2, 5, 7, 8, 11, 12 };
```

```
int tam = numeros.length; // tam == 6
int[][] matriz = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
int tam2 = matriz.length; // tam == 3
```

4.4.10.8 Array.add()

Añade un elemento al final de la lista o array de una dimensión.

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}
```

4.4.11 Método Main

Este es el método donde se iniciará la ejecución de Javalang. Puede estar declarado en cualquier parte del archivo.

Sintaxis:

public static void main() {

<sentencias>

}

4.5 Especificación de la generación de código ARM64

La generación de código assembler para la arquitectura ARM64 implica la traducción de las estructuras del lenguaje JavaLang instrucciones específicas del conjunto de instrucciones ARM64. Durante este proceso, se mapean las construcciones del lenguaje, como declaraciones de variables, expresiones aritméticas, control de flujo y llamadas a funciones, a secuencias de instrucciones ARM64 que implementan la funcionalidad deseada.

Para la generación de código assembler en la arquitectura ARM64, se optará por utilizar el conjunto de instrucciones AArch64, que representa la implementación de 64 bits de la arquitectura ARM. Este conjunto de instrucciones proporciona operaciones fundamentales de aritmética, lógica y transferencia de datos, esenciales para la ejecución de programas en JavaLang.

Para verificar la salida del compilador, se utilizará **QEMU**, un emulador y virtualizador de código abierto que permite ejecutar código en la arquitectura ARM64 emulando un procesador **Cortex-A57**. Esto permitirá probar el código sin necesidad de hardware físico. En este caso, QEMU proporcionará un entorno adecuado para ejecutar el código ensamblador generado y evaluar su comportamiento.

Alternativamente, el código ensamblador generado podrá ejecutarse directamente en hardware compatible, como una **Raspberry Pi** con un procesador ARM64. **Ambas opciones, ya sea virtualizando con QEMU o ejecutando el código directamente sobre un dispositivo físico compatible como lo es la Raspberry Pi**, son válidas para verificar la traducción del código generado.

Por lo tanto, el compilador debe generar un archivo con extensión **".s"** que contenga el código ensamblador producido. Este archivo se utilizará para validar el código tanto en el simulador QEMU o en una Raspberry Pi, permitiendo ejecutar el código en un entorno simulado o en hardware real para verificar su correcto funcionamiento.

- Para poder utilizar el emulador de Qemu, tomar como referencia la siguiente documentación: https://mariokartwii.com/armv8/ch5.html
- Para acceder a la documentación oficial de ARM64 utilizar el siguiente enlace: https://mariokartwii.com/armv8/

4.5.1 Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis de la entrada. Se utilizará el formato del mismo ARM64 para comentarios:

- Los comentarios de una línea serán delimitados al inicio con los símbolos “#” y al final con un carácter de finalización de línea.
- Los comentarios con múltiples líneas empezarán con los símbolos “*/” y terminarán con los símbolos “*/*”.

```
# Este es un comentario en ARM64
# La siguiente línea carga el valor 10 en el registro x10
mov x10, #10
/* Esto es un comentario multilínea */
```

En este ejemplo, las líneas que comienzan con # son comentarios y no tienen ningún impacto en el código ensamblador. Los comentarios son útiles para explicar el código y hacer anotaciones.

4.5.2 Registros

En ARM64, los tipos de datos se manejan principalmente a nivel de registros y direcciones de memoria, sin tipos de datos explícitos como en lenguajes de más alto nivel. Los registros almacenan diferentes tipos de datos dependiendo de su propósito.

4.5.2.1 Registros de propósito general

Estos registros pueden usarse para almacenar enteros, direcciones de memoria o cualquier otro tipo de dato representable en el tamaño del registro.

```
# Cargar un entero en el registro x10
mov x10, #42

# Cargar una dirección de memoria en el registro x11
adr x11, mi_etiqueta
```

4.5.2.2 Registros de coma flotante

Estos registros se utilizan para operaciones de punto flotante, soportando números en formatos de precisión simple o doble, de acuerdo con el estándar IEEE-754.

```
# Cargar un número de punto flotante de precisión simple
# en el registro s10
ldr s10, [mi_etiqueta]

# Cargar un número de punto flotante de doble precisión
# en el registro d11
ldr d11, [mi_etiqueta_doble]
```

4.5.2.3 Memoria

La memoria en ARM64 se maneja como un conjunto de direcciones que apuntan a bytes. No existe un tipo de dato explícito para la memoria, pero se puede almacenar y cargar datos en direcciones específicas de memoria.

```
# Almacenar un valor en la memoria
```

```
str x10, [x11]   # Almacena el valor de x10 en la dirección apuntada por x11

# Cargar un valor de la memoria en un registro
ldr x12, [x11]   # Carga el valor de la dirección apuntada por x11 en x12
```

4.5.2.4 Etiquetas

En ARM64, las etiquetas también se utilizan para indicar ubicaciones específicas dentro del código y permitir saltos (saltos condicionales o incondicionales) durante la ejecución. Las etiquetas se definen de manera similar a como se hace en otros lenguajes ensambladores, y se utilizan en combinación con instrucciones de salto.

Las etiquetas se definen en el código como identificadores únicos seguidos de dos puntos (:). Estas se utilizan para marcar ubicaciones en el código y se utilizan junto con instrucciones de salto como b (salto incondicional) o bne (salto condicional) para mover el flujo de ejecución.

Las etiquetas no requieren un formato específico, pero deben ser identificadores válidos. En ARM64, estas etiquetas se usan para referirse a ubicaciones en el código y permitir saltos a esas ubicaciones.

```
L1:
    # Código para inicializar un bucle
    mov x10, #0    # Inicializa el contador en 0
    mov x11, #10   # Inicializa el valor de salida del bucle

L3:
    # Código del bucle
    add x10, x10, #1   # Incrementa el contador en 1

    # Comprobación de la condición de salida del bucle
    cmp x10, x11        # Compara x10 con x11
    bne L3               # Salta al inicio del bucle si x10 no es igual a
    x11

L100:
    # Código después del bucle
    mov x12, #0    # Código posterior al bucle
```

Este ejemplo muestra cómo utilizar etiquetas para controlar el flujo de ejecución dentro de un bucle en ARM64.

4.5.2.5 Saltos

En ARM64, las instrucciones de salto condicional e incondicional funcionan de manera similar a otros ensambladores, permitiendo cambiar el flujo de ejecución del programa.

Instrucciones de salto condicional:
Las instrucciones de salto condicional permiten realizar saltos basados en el estado de los registros, como resultados de comparaciones previas. ARM64 tiene una serie de instrucciones para realizar saltos condicionales dependiendo de los indicadores de estado (como Zero, Negative, Carry, y Overflow) almacenados en el registro NZCV.

```
# Saltar a la etiqueta L1 si x2 es igual a x3
cmp x2, x3      # Compara x2 con x3
b.eq L1         # Salta a L1 si x2 es igual a x3

# Saltar a la etiqueta L2 si x4 no es igual a x5
cmp x4, x5      # Compara x4 con x5
b.ne L2         # Salta a L2 si x4 no es igual a x5

# Saltar a la etiqueta L3 si x6 es menor que x7
cmp x6, x7      # Compara x6 con x7
b.lt L3         # Salta a L3 si x6 es menor que x7
```

Instrucciones de salto incondicional

Estas instrucciones siempre realizan el salto, sin ninguna condición asociada.

```
# Saltar incondicionalmente a la etiqueta L1
b L1            # Salta a L1

# Saltar incondicionalmente a la etiqueta L2 y guardar la dirección de
 retorno
bl L2           # Salta a L2 y guarda la dirección de retorno en el registro
 x30 (Link Register)
```

```
# Saltar a la dirección almacenada en el registro x9
br x9           # Salta a la dirección especificada en el registro x9
```

4.5.2.6 Operadores

En ARM64, las operaciones aritméticas básicas se realizan mediante instrucciones específicas, similares a las de otros lenguajes ensambladores. A continuación se detallan cómo realizar las operaciones básicas de suma, resta, multiplicación, división, y operaciones de incremento y decremento en ARM64.

Suma:

```
add x3, x1, x2   # x3 = x1 + x2
```

Resta:

```
sub x3, x1, x2   # x3 = x1 - x2
```

Multiplicación:

```
mul x3, x1, x2   # x3 = x1 * x2
```

División:

ARM64 tiene instrucciones para división de enteros: sdiv (división con signo) y udiv (división sin signo)

```
sdiv x3, x1, x2  # x3 = x1 / x2 (división con signo)
```

Incremento y decremento:

```
subi x3, x3, 1   # Decrementa x3 en 1
addi x3, x3, 1   # Incrementa x3 en 1
```

4.6 Entregables

Tipo	Descripción
Código fuente	Repositorio Git organizado (GitHub/GitLab) con todo el código C.
Informe técnico	Documento (máx. 10 págs.) que describa: • Gramática formal de JavaLang

	● Métodos utilizados para generar código a bajo nivel. ● Decisiones de diseño y desafíos enfrentados

- **Prototipo funcional**
  - Código fuente completo en el repositorio.

- **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 adaptar el tiempo de trabajo en bloques de 1–2 semanas.

1. **Investigación y Preparación**

- Revisión de conceptos avanzados de teoría de compiladores relevantes para la traducción a código máquina, incluyendo:
    - Representaciones intermedias (IR)
    - Manejo de pila y registros
    - Convenciones de llamada (calling conventions)
- Revisión de ejemplos de compiladores dirigidos por sintaxis implementados en Java, con enfoque en la construcción de AST y generación de código.

2. **Diseño de la Gramática y Arquitectura**
    - Definir la gramática formal de JavaLang (.usl), cubriendo:
        - Variables, estructuras de control, funciones, arreglos y expresiones
        - Sin incluir elementos de orientación a objetos
    - Diseñar la arquitectura modular del compilador, con al menos los siguientes paquetes:
        - Lexer, parser, ast, semantic, codegen, y cli (si se requiere una interfaz por consola)
    - Esquematizar el flujo de procesamiento desde el código fuente hasta la generación del archivo ensamblador ARM64.
3. **Desarrollo Iterativo**

Bloque 1:

- Planear la reutilización de la gramática y elementos de código del interprete realizado en el proyecto 1.

Bloque 2:

- Construir el Árbol de Sintaxis Abstracta (AST).

- Diseñar e implementar el análisis semántico, incluyendo la tabla de símbolos y validaciones de tipo.

Bloque 3 y 4:

- Implementar la traducción del AST o IR a código ensamblador ARM64.
- Generar archivos `.s` listos para ser ensamblados con `as` o simulados con `qemu-aarch64`.

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	23/09/2025	23/09/2025
Elaboración	23/09/2025	24/10/2025
Calificación	25/10/2025	29/10/2025

8 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.

8.1 Comentarios Generales

_____

_____

_____

_____