



Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

**Fase 2**

Isaac Ramírez Rojas

Adrian José Villalobos Peraza

Compiladores e Interpretes

Verano 2024-2025

<b>Manual de Usuario.....</b>	<b>3</b>
Instrucciones de compilación.....	3
Ejecución.....	4
Uso.....	5
<b>Pruebas de Funcionalidad.....</b>	<b>6</b>
<b>Descripción de Problema.....</b>	<b>7</b>
<b>Diseño del Programa.....</b>	<b>8</b>
Decisiones de Diseño.....	8
Algoritmos Usados.....	10
<b>Librerías Usadas.....</b>	<b>10</b>
<b>Análisis de Resultados.....</b>	<b>11</b>
<b>Bitácora.....</b>	<b>14</b>

# Manual de Usuario

## Instrucciones de compilación

Para llevar a cabo la compilación del proyecto es indispensable contar con las siguientes herramientas y librerías:

- **Java (JDK):** Asegúrese de tener instalado el Java Development Kit en una versión adecuada (por ejemplo, la versión 8 o superior). Las pruebas realizadas a este proyecto fueron con Oracle OpenJDK 23.0.1
- **Gradle:** Esta herramienta de automatización de construcción facilitará la resolución de dependencias, la ejecución de tareas de compilación y la empaquetación del proyecto. Adicionalmente, se puede utilizar en la línea de comandos para la ejecución del proyecto.
- **JavaCup:** Esta librería se utiliza para generar el analizador sintáctico (parser). A partir de la gramática definida, JavaCup se encarga de producir el código Java que valida la estructura sintáctica del programa fuente. En este caso es requerida únicamente para proporcionar los símbolos requeridos por el lexer, por medio del archivo `sym.java`.
- **JFlex:** JFlex permite generar el analizador léxico (lexer), el cual convierte el flujo de caracteres del código fuente en tokens. Se utiliza para producir el `Lexer.java`.

Antes de iniciar la compilación, es importante revisar la estructura de archivos del proyecto. Asegúrese de que el archivo `Lexer.java`, así como `parser.java` y `sym.java` (generados a partir de las especificaciones de JFlex y JavaCup), se encuentren en la carpeta `parser`. Esto es estrictamente necesario.

En cuanto al entorno de desarrollo, las pruebas realizadas se llevaron a cabo utilizando IntelliJ IDEA de JetBrains. Esta herramienta integrada permite compilar, ejecutar y depurar el proyecto de forma intuitiva. Si ha optado por utilizar IntelliJ, puede ejecutar el proceso de compilación directamente desde su interfaz, de lo contrario puede ser por línea de comandos con gradle. Cabe destacar que el proyecto fue ejecutado y probado en entornos Linux y MacOS, se desconoce su funcionamiento en sistemas operativos Windows.

## Ejecución

La ejecución del proyecto puede llevarse a cabo desde el entorno de desarrollo integrado IntelliJ IDEA. Una vez que se hayan satisfecho todas las dependencias, completado el proceso de compilación y ubicado los archivos generados en las carpetas correspondientes, es posible ejecutar el programa con el botón dedicado a esta acción.

Una vez ya realizados los pasos anteriores se procede a realizar la ejecución de programa, cuando ejecutamos el programa nos van salir en terminal 3 opciones, una para generar los archivos necesarios, en este caso primero debemos seleccionar esta opción, una vez ya se generaron, un script automáticamente moverá los archivos a una carpeta denominada parser, en esta carpeta se encuentran los 3 archivos más importantes `Lexer.java`, `parser.java`, `sym.java`. Una vez que se complete esta etapa, procedemos a hacer la etapa dos, en la cuál ya se evalúa el análisis sintáctico, donde se mostrarán errores respectivos cuando se ejecuta el programa, además se mostrará la tabla de símbolos.

Un aspecto importante a aclarar es que la **tabla de símbolos** se muestra **por ámbitos**. Esto significa que cada vez que se inicia un nuevo bloque de código (por ejemplo, al entrar en una función, en un `if`, `while`, etc.), se crea un nuevo **ámbito**. Cada ámbito posee sus propias variables locales que solo son válidas dentro de ese bloque. Al imprimir la tabla de símbolos, se listan los ámbitos en el orden en que fueron creados, por lo que verás una estructura como “Ámbito 0, Ámbito 1, Ámbito 2, ...”. Cada ámbito se muestra con las variables que se declararon en ese nivel concreto, junto con su tipo y posición en el código fuente.

```
int _func(int _a) {
    char _variabledefunc1_;
    if (_a > 0) {
        boolean _varbool_;
        // ...
    }
    // ...
}

=== Tabla de símbolos de función '_func_' ===

Ámbito 0: []
Ámbito 1: [_variabledefunc1_: char (2, 9)]
Ámbito 2: [_varbool_: boolean (4, 16)]
```

- **Ámbito 0** se crea al iniciar la función (no hay variables declaradas directamente ahí).
- **Ámbito 1** corresponde al bloque principal dentro de la función, donde se declara `_variabledefunc1_`.
- **Ámbito 2** se genera al entrar en el bloque `if`, donde se declara `_varbool_`.

## Uso

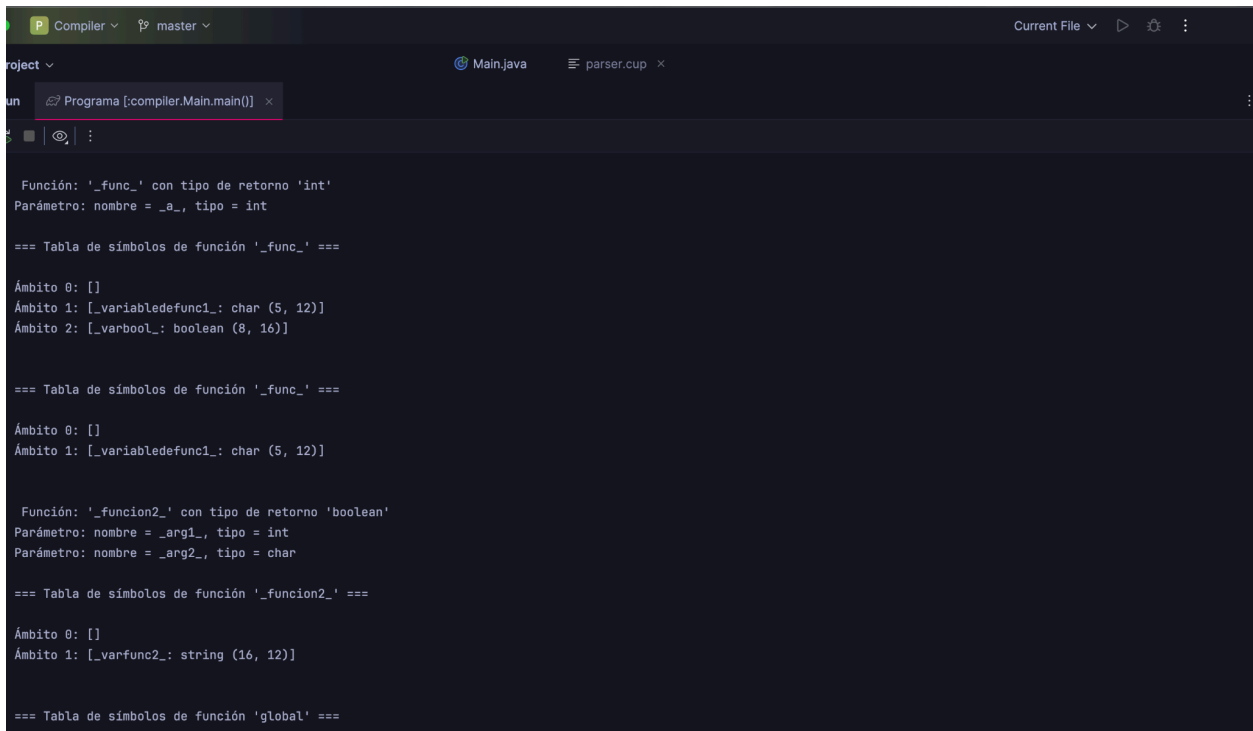
El programa está diseñado para procesar archivos de texto y realizar tanto un **análisis léxico** como un **análisis sintáctico** sobre su contenido, proporcionando información estructurada acerca de los elementos del lenguaje presentes en dicho archivo. Para usar el programa, el usuario debe ingresar algunas de las opciones disponibles (del 1 al 3: generar archivos necesarios, probar el lexer o salir). En caso de seleccionar la segunda opción, el usuario únicamente debe proporcionar un archivo de texto válido, cuya ruta deberá ser especificada como entrada en el momento de la ejecución (esta ruta debe ser fiel al root del proyecto).

Una vez proporcionado el archivo, el programa procede a abrirlo y leer su contenido línea por línea. A medida que procesa el texto, realiza un **análisis léxico** (construido con *JFlex*), identificando y clasificando los componentes del lenguaje (palabras clave, identificadores, operadores, literales, etc.) según las reglas definidas. Posteriormente, se aplica el **análisis sintáctico** (implementado con *CUP*), el cual valida la estructura y las relaciones entre los tokens generados. En caso de que existan errores en la secuencia de tokens (por ejemplo, un identificador mal ubicado o una estructura inválida), el programa muestra mensajes de error que indican la línea y la columna donde se detectó el problema, para facilitar su corrección.

Todo este proceso se imprime en la consola, ofreciendo retroalimentación inmediata sobre el contenido del archivo. Además, se genera un archivo de salida (`salida.txt`) en el directorio `src/lex`, que contiene la información correspondiente al análisis léxico (la lista de tokens reconocidos). Es importante destacar que los mensajes de error (tanto léxicos como sintácticos) se muestran únicamente en la consola y no se incluyen en el archivo de salida. De

esta forma, el usuario puede revisar de forma separada el listado de tokens generados y cualquier posible mensaje de error detectado durante el análisis.

## Pruebas de Funcionalidad



```
Project: Compiler master
Main.java parser.cup
Programa [compiler.Main.main()] x

Función: '_func_' con tipo de retorno 'int'
Parámetro: nombre = _a_, tipo = int

=== Tabla de símbolos de función '_func_' ===

Ámbito 0: []
Ámbito 1: [_variabledefunc1_: char (5, 12)]
Ámbito 2: [_varbool_: boolean (8, 16)]

=== Tabla de símbolos de función '_func_' ===

Ámbito 0: []
Ámbito 1: [_variabledefunc1_: char (5, 12)]

Función: '_funcion2_' con tipo de retorno 'boolean'
Parámetro: nombre = _arg1_, tipo = int
Parámetro: nombre = _arg2_, tipo = char

=== Tabla de símbolos de función '_funcion2_' ===

Ámbito 0: []
Ámbito 1: [_varfunc2_: string (16, 12)]

=== Tabla de símbolos de función 'global' ===
```

```

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]
Ámbito 3: [_z_: int (31, 21)]
Ámbito 4: [_w_: int (36, 25)]

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]
Ámbito 3: [_z_: int (31, 21)]
Ámbito 4: [_t_: int (42, 25)]

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]
Ámbito 3: [_z_: int (31, 21)]
Ámbito 4: []

instead expected token classes are []

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_v_: int (38, 17)]

```

```

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]
Ámbito 3: [_z_: int (31, 21)]

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]
Ámbito 3: []

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]
Ámbito 2: [_x_: int (29, 17)]

=== Tabla de símbolos de función 'global' ===

Ámbito 0: []
Ámbito 1: [_y_: int (21, 13), _a_: int (22, 13)]

Análisis sintáctico completado correctamente.

BUILD SUCCESSFUL in 9s

```

## Descripción de Problema

El problema consiste en la creación de un analizador sintáctico, el cual trabajará en conjunto con el analizador léxico generado en el proyecto anterior. El objetivo es recibir un archivo fuente y analizar este en cuanto a sintaxis y léxico.

El análisis sintáctico tiene como objetivo validar que las secuencias de tokens (identificadas durante el análisis léxico) cumplan con las reglas gramaticales del lenguaje. Para ello, se emplea un conjunto de producciones que definen cómo se deben agrupar y ordenar los tokens, asegurando que el código fuente tenga una estructura coherente. Mientras que el analizador léxico sigue regido por las indicaciones del proyecto anterior.

Se debe utilizar CUP para construir el parser a partir de la gramática definida. Durante el proceso de análisis, el parser debe ir reconociendo cada construcción sintáctica y, cuando encuentra una secuencia de tokens que no se ajusta a la gramática, generar un error sintáctico con información relevante (línea y columna) para facilitar su localización y corrección. Adicionalmente, se debe construir una tabla de símbolos, donde se registran cada una de las variables y funciones definidas, organizadas por ámbitos (o *scopes*) a medida que estos van apareciendo en el código.

La gestión de errores debe seguir el modo de recuperación en modo pánico, es decir, reportar el error y seguir con la ejecución y análisis del archivo fuente. Cabe destacar que se deben mantener todas las funcionalidades del proyecto uno.

## Diseño del Programa

### Decisiones de Diseño

Uno de los principales desafíos en el desarrollo de este proyecto fue la gestión efectiva de los **scopes** dentro de la **tabla de símbolos**, así como el manejo robusto de errores para asegurar que el programa pueda reportarlos sin fallos.

Para abordar la **gestión de scopes**, se implementó una estructura de **pila (stack)** como componente global dentro de la clase `SymbolTable`. Esta clase incluye una subclase denominada `FunctionInfo`, que contiene dos propiedades esenciales:

1. **Stack de `SymbolInfo`**: Representa los diferentes ámbitos (scopes) anidados dentro de una función. Cada vez que se entra en un nuevo scope, se realiza un **push** a la pila, y al salir de dicho scope, se ejecuta un **pop** para eliminarlo. Esta estructura permite manejar



los scopes de manera **LIFO (Last In, First Out)**, lo que refleja el comportamiento típico de los lenguajes de programación al gestionar bloques anidados.

2. **Lista de `SymbolInfo`:** Almacena los parámetros de la función correspondiente. Esta lista facilita el acceso rápido a los parámetros cuando se realiza la búsqueda de símbolos dentro del scope actual.

La decisión de utilizar una pila para manejar los scopes se fundamenta en la necesidad de declarar un nuevo ámbito para cada bloque dentro de una función, similar a cómo los lenguajes de programación gestionan los bloques de código. Al apilar los scopes, se garantiza que siempre se maneje el scope más reciente, y al realizar un `pop`, se elimina de manera eficiente el scope que ha finalizado.

- **Manejo de Errores:** El manejo de errores se implementó mediante la integración de mecanismos específicos en el archivo `.cup`. Se definieron métodos sobrescritos como `report_error`, `report_fatal_error` y `unrecovered_syntax_error` dentro de la sección `parser code` para capturar y reportar errores sintácticos durante el análisis. Estos métodos verifican si la información proporcionada es una instancia de `Symbol` y, de ser así, extraen detalles como la línea y columna donde ocurrió el error, así como el valor del símbolo involucrado. Por ejemplo, `report_error` y `report_fatal_error` imprimen mensajes de error detallados que incluyen la ubicación precisa del problema. Además, se implementaron reglas de producción específicas, como `encontrar_error` y `encontrar_error_aux`, para manejar errores no recuperables y permitir la recuperación del análisis sintáctico sin detener completamente el proceso.
- **Impresión de la Tabla de Símbolos:** Por otro lado, la impresión de la tabla de símbolos se realiza una vez que el parser ha completado el análisis sintáctico del contenido. Este proceso garantiza que solo se muestran los scopes activos y sus respectivos símbolos, manteniendo una representación clara y precisa de los ámbitos presentes en el código analizado. De esta manera, se evita la impresión de ámbitos vacíos y se optimiza la presentación de la información relevante.

Es importante destacar que cuando una variable sale de su scope, se elimina automáticamente del ámbito correspondiente. Si en el intervalo entre la eliminación de una variable y la declaración de una nueva variable en el mismo scope se realiza una nueva declaración, la nueva variable reemplaza a la anterior en la impresión de la tabla de símbolos.

Esto evita la impresión de ámbitos vacíos y reduce la cantidad de operaciones de impresión, optimizando así el rendimiento del programa y manteniendo la coherencia en la representación de los scopes.

## Algoritmos Usados

En este proyecto no se desarrollaron algoritmos personalizados, ya que se aprovechó la infraestructura proporcionada por las herramientas JFlex y CUP. Basándose en las documentaciones y guías de estas herramientas, se implementó el análisis sintáctico(parser) necesarios para el procesamiento del lenguaje.

## Librerías Usadas

Las librerías usadas no difieren con las del proyecto anterior, son exactamente las mismas. Se describen a continuación.

**Librerías de manejo de archivos:** Estas librerías se utilizan para la gestión y organización de los archivos generados durante el proceso de construcción del compilador, en particular aquellos producidos por JFlex y JavaCup. Por ejemplo, una vez que JFlex y JavaCup han generado los archivos `Lexer.java`, `parser.java` y `sym.java`, se emplean funcionalidades de `java.nio.file` y manejo de excepciones `IOException` para mover, reubicar o sobrescribir estos ficheros en las rutas correspondientes del proyecto, asegurando así una correcta estructura interna del compilador.

**JavaCup:** JavaCup es una herramienta que se utiliza para generar el analizador sintáctico, la siguiente etapa después del análisis léxico. Una vez que JFlex ha proporcionado la secuencia de tokens, el parser generado por JavaCup toma estos tokens y los combina de acuerdo con las reglas gramaticales definidas en el archivo de especificaciones (`.cup`). Este archivo describe la gramática en notación BNF o similar, detallando cómo los tokens deben organizarse para conformar sentencias y estructuras sintácticas válidas del lenguaje. El objetivo principal de JavaCup es verificar si la secuencia de tokens cumple con las reglas de la gramática, sin embargo, como se mencionó anteriormente, en este proyecto su uso se reduce al proporcionamiento de los símbolos requeridos. A diferencia del proyecto anterior, esta vez JavaCup sirvió como librería para generar una gramática y manejar errores sintácticos.

**Jflex:** JFlex es una herramienta que automatiza la creación del analizador léxico, es decir, se encarga de traducir el texto fuente del programa en una secuencia de tokens. Un token representa la unidad mínima de significado en un lenguaje de programación (por ejemplo, palabras clave, identificadores, literales, símbolos de operación, etc.). JFlex toma como entrada un archivo de especificaciones (normalmente con extensión .jflex) donde se definen las expresiones regulares que describen cada tipo de token. A partir de estas especificaciones, JFlex genera una clase Java (Lexer.java) que, al ejecutarse, lee el código fuente carácter por carácter y crea la lista ordenada de tokens para su posterior análisis. En el caso de este proyecto, los cambios en el JFlex fueron solo para corregir aspectos del proyecto anterior, sin embargo, sigue cumpliendo la misma función.

## Análisis de Resultados

Objetivo	Estado	Comentario
Crear una gramática que se acople a las instrucciones requeridas del programa (Un programa escrito para este lenguaje está compuesto por una secuencia de declaraciones de procedimientos, que contienen diferentes expresiones y asignaciones de expresiones; todo programa debe contener un único método main.	Completado	

<p>Leer un archivo fuente y de este decir todos sus tokens, errores léxicos, sintácticos e imprimir información de la tabla de símbolos</p>	<p>Completado</p>	<p>La tabla de símbolos (el scope actual, no la tabla en general) se imprime cada vez que termina el scope actual, por lo que un scope puede ser impreso varias veces en consola si presenta otros ámbitos anidados en él.</p>
<p>Por cada token deberán indicar en cuál tabla de símbolos va y cual información se almacenará (gestión de tablas de símbolos).</p>	<p>Completado</p>	
<p>Reportar y manejar los errores léxicos y sintácticos encontrados. Debe utilizar la técnica de Recuperación en Modo Pánico (error en línea y continúa con la siguiente).</p>	<p>Completado</p>	<p>Los errores se manejan con el no terminal reservado de cup "error". Para esto se da uso de ese no terminal seguido de un token de recuperación, como un punto y coma. Dependiendo del error y si hay errores en la misma línea, es posible que no se detecten todos ellos. Para esto, se deben ir corrigiendo y a como se corrijan se irán mostrando más errores subsecuentes.</p>

Generar un archivo de salida que contiene la información de los tokens encontrados (nombre, línea y columna, parte del proyecto anterior).	Completado	
Crear un menú como interfaz de usuario, el cual tenga la opción de generar los archivos del compilador, hacer uso de él o salir (parte del proyecto anterior).	Completado	
Utilizar GitHub como gestor de versiones.	Completado	<a href="https://github.com/akhos/Christmas-Compiler">https://github.com/akhos/Christmas-Compiler</a> . Existen distintas ramas para cada fase.
Documentar internamente el código, indicando que recibe, que hace y devuelve una función o método	Completado	Se utilizó la directiva Javadoc para la documentación interna del proyecto.
Corregir aspectos del proyecto uno en base a la retroalimentación del profesor.	Completado	Se corrigieron aspectos del analizador léxico en base a la retroalimentación del profesor.

# Bitácora

En esta sección se encuentran todos los commits realizados en el repositorio, presentados de manera estructurada y descriptiva para su fácil consulta.

Los estudiantes son:

- iZack\_26: Isaac Ramirez
- akhoz: Adrián Villalobos



URL del repositorio: [GitHub](#)