

Instituto Tecnológico de Costa Rica

IC5701 Compiladores e intérpretes

Grupo: 60

2025, Semestre I

Profesor: Allan Rodríguez

Estudiantes:

Andrew López Herrera – 2021062132

Hillary Malespín Ulloa – 2021106074

Asignación:

Proyecto 02 Análisis semántico y código intermedio

Entrega: lunes 02 de junio, 2025

Portada.....	2
Manual de usuario: instrucciones de compilación, ejecución y uso.....	2
Pruebas de funcionalidad	5
Descripción del problema.....	8
Diseño del programa: decisiones de diseño, algoritmos usados.....	9
Librerías usadas:.....	11
Ánalisis de resultados	11
Bitácora (autogenerada en git, commit por usuario incluyendo comentario).....	12

Manual de usuario: instrucciones de compilación, ejecución y uso

Los siguientes archivos se encuentran en la carpeta
programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer del proyecto.

1. Se debe compilar el archivo `Lexer.flex` con el comando `jflex Lexer.flex`.

```
PS C:\TEC\IS2025\Compiladores e Interpretes\PY01\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer> jflex Lexer.flex
Reading "Lexer.flex"
Constructing NFA : 456 states in NFA
Converting NFA to DFA :
.....
210 states before minimization, 191 states in minimized DFA
Old file "Milexer.java" saved as "Milexer.java~"
Writing code to "Milexer.java"
PS C:\TEC\IS2025\Compiladores e Interpretes\PY01\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer>
```

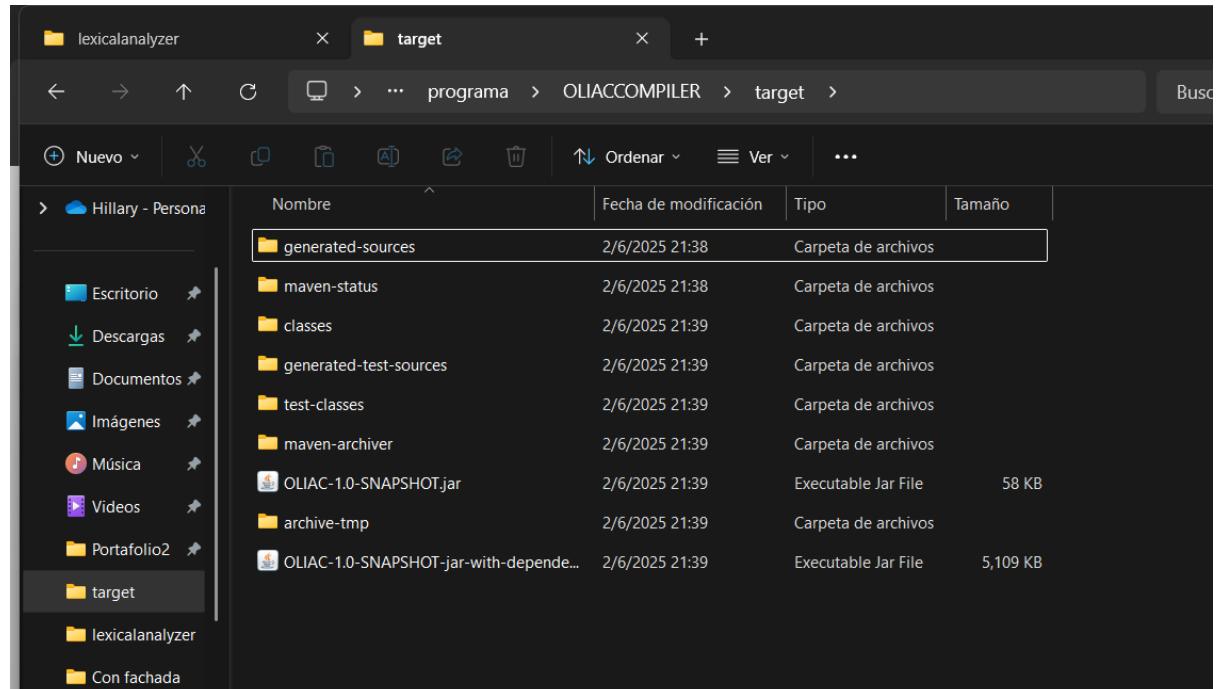
2. Se debe compilar el archivo `Parser.cup` con el comando `java -jar java-cup-11b.jar -parser parser -symbols sym Parser.cup`

```
PS C:\TEC\IS2025\Compiladores e Interpretes\PY01\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\src\main\java\com\lexicalanalyzer> java -jar java-cup-11b.jar -parser parser -symbols sym Parser.cup
----- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -----
0 errors and 0 warnings
63 terminals, 50 non-terminals, and 129 productions declared,
producing 233 unique parse states.
0 terminals declared but not used.
0 non-terminals declared but not used.
0 productions never reduced.
0 conflicts detected (0 expected).
Code written to "parser.java", and "sym.java".
----- (CUP v0.11b 20160615 (GIT 4ac7450))
```

3. Se debe compilar el proyecto en NetBeans utilizando Clean and Build para generar un nuevo proyecto.



4. Se debe ir a la carpeta programa\OLIACCOMPILER\target del proyecto. Ahí se encontrará el archivo OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar que es el proyecto compilado. Ahora solo hay que ingresar un archivo de pruebas.

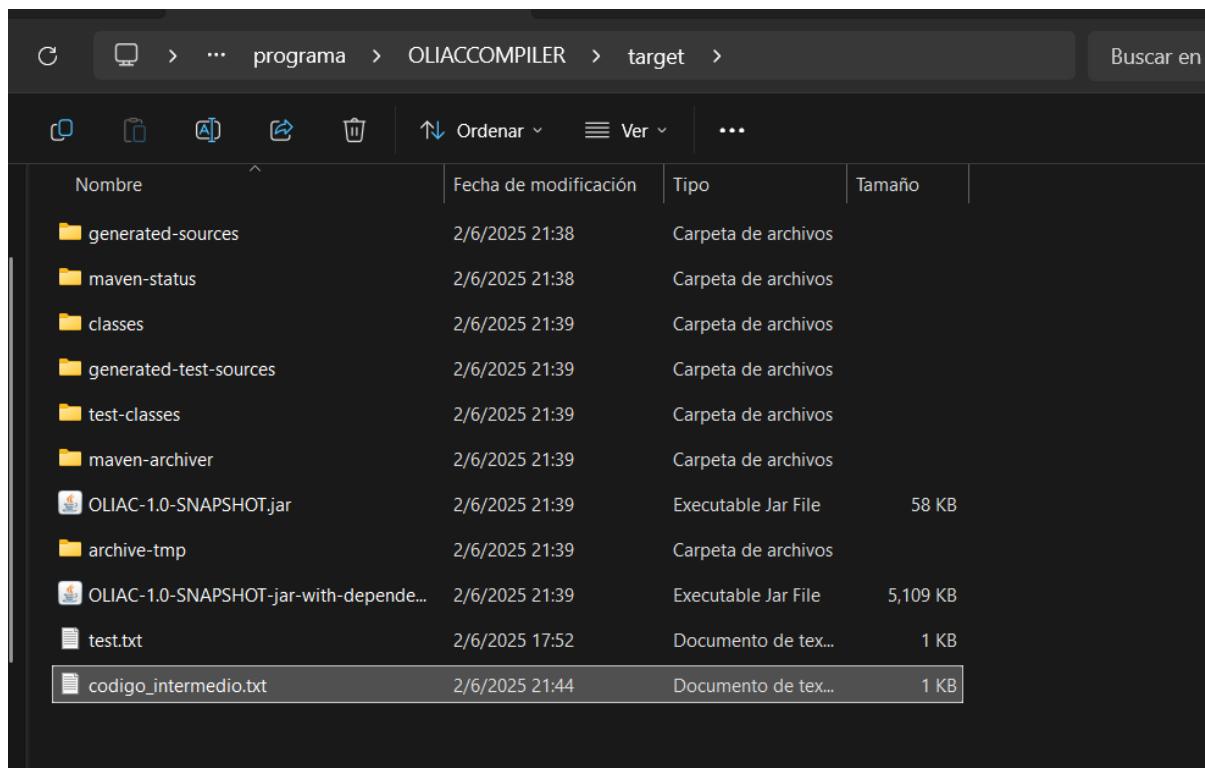


5. Cuando se haya agregado el archivo de prueba se puede utilizar el comando **java -jar OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar -f pruebal.txt** para realizar el

análisis léxico, se puede cambiar prueba1.txt por cualquier nombre de archivo existente.

```
PS C:\TEC\IS2025\Compiladores e Interpretes\PY01\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\target> java -jar OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar -f "test.txt"
--- Parseo exitoso, sin errores sintácticos ---
--- Análisis semántico exitoso ---
--- Generando código intermedio en: codigo_intermedio.txt ---
PS C:\TEC\IS2025\Compiladores e Interpretes\PY01\IProyectoCompiladoresEInterpretes\programa\OLIACCOMPILER\target> |
```

6. Si el archivo no presenta errores se genera el archivo de código intermedio que se va a guardar en la misma carpeta programa\OLIACCOMPILER\target.



Nombre	Fecha de modificación	Tipo	Tamaño
generated-sources	2/6/2025 21:38	Carpeta de archivos	
maven-status	2/6/2025 21:38	Carpeta de archivos	
classes	2/6/2025 21:39	Carpeta de archivos	
generated-test-sources	2/6/2025 21:39	Carpeta de archivos	
test-classes	2/6/2025 21:39	Carpeta de archivos	
maven-archiver	2/6/2025 21:39	Carpeta de archivos	
OLIAC-1.0-SNAPSHOT.jar	2/6/2025 21:39	Executable Jar File	58 KB
archive-tmp	2/6/2025 21:39	Carpeta de archivos	
OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar	2/6/2025 21:39	Executable Jar File	5,109 KB
test.txt	2/6/2025 17:52	Documento de tex...	1 KB
codigo_intermedio.txt	2/6/2025 21:44	Documento de tex...	1 KB

Pruebas de funcionalidad

Pruebas de análisis semántico:

Prueba Uno:

```

float obtenerNotaFinal| int nota1, int nota2, int nota3 | \
    int suma = nota1 + nota2 + nota3?
    float promedio = suma // 3?
    return promedio?
/

string determinarEstado| float notaFinal | \
    if|notaFinal >= 90| \
        return "Excelente"?
    / 
    elif|notaFinal >= 70| \
        return "Aprobado"?
    / 
    else \
        return "Reprobado"?
/
 

string mostrarEstudiante| string nombre, float notaFinal | \
    writeString -> "Nombre del estudiante:"?
    writeString -> nombre?
    writeString -> "Nota final:"?
    writeFloat -> notaFinal?
    string estado = determinarEstado|notaFinal|?
    writeString -> "Estado:"?
    writeString -> estado?
/
 

void main| | \
    int cantidadEstudiantes = 2?
    int i = 0?
    int nombres[2] = |2, 3|?
    int notasFinales[2] = |5, 100000000|?

    for|i = 0? i < cantidadEstudiantes? ++i| \
        writeString -> "Ingrese las 3 notas del estudiante:"?
        int n1 = 0?
        int n2 = 0?
        int n3 = 0?
        readInt <- n1?
        readInt <- n2?
        readInt <- n3?
        float nf = obtenerNotaFinal|n1, n2, n3|?
        notasFinales[i] = nf?
    /
 

    writeString -> "Mostrando resultados:"?
    i = 0?
    for|i = 0? i < cantidadEstudiantes? ++i| \
        string nombre = "nombres[i]"?
        float nota = 7.056845105?
        mostrarEstudiante|nombre, nota|?
    /
 

    writeString -> "Fin del programa."?
/

```

```

PS E:\GitHub\Compiladores\IProyectoCompiladoresEInterpretes\programa\OLIACCO
MPILER\target> java -jar OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar -f "ej
emplo2.c"

--- Parseo exitoso, sin errores sintácticos ---
--- Análisis semántico exitoso ---
--- Generando código intermedio en: codigo_intermedio.txt ---

```

Se puede observar que todo el código es procesado sin problemas.

Prueba 2

```

1 float obtenerNotaFinalJ int nota1, int nota2, int nota3 l \
2     int suma = nota1 + nota2 + nota3?
3     float promedio = suma // 3?
4     return promedio?
5 /
6
7 string determinarEstadoJ float notaFinal l \
8     if notaFinal >= 90l \
9         return "Excelente"?
10    /
11    elif notaFinal >= 70l \
12        return "Aprobado"?
13    /
14    else \
15        return notaFinal?
16    /
17 /
18
19 string mostrarEstudianteJ string nombre, float notaFinal l \
20     writeString -> "Nombre del estudiante:"?
21     writeString -> nombre?
22     writeString -> "Nota final:"?
23     writeFloat -> notaFinal?
24     string estado = determinarEstadoJnotaFinal?
25     writeString -> "Estado:"?
26     writeString -> estado?
27 /
28
29 void mainJl \
30     int cantidadEstudiantes = 2?
31     int i = 0?
32     int nombres[2] = |2, 3|?
33     int notasFinales[2] = |5, 1000000000|?
34
35     forJi = 0? i < cantidadEstudiantes? ++i] \
36         writeString -> "Ingrese las 3 notas del estudiante:"?
37         int n1 = 0?
38         int n2 = 0?
39         int n3 = 0?
40         readInt <- n1?
41         readInt <- n2?
42         readInt <- n3?
43         float nf = obtenerNotaFinalJn1, n2, n3l?
44         notasFinales[i] = n3?
45     /
46
47     writeString -> "Mostrando resultados:"?
48     i = 0?
49     forJi = 0? i < cantidadEstudiantes? ++i] \
50         string nombre = "nombres[i]"?
51         float nota = 7.056845105?
52         mostrarEstudianteJnombre, nota?
53     /
54
55     writeString -> "Fin del programa."?
56
57

```

PS E:\GitHub\Compiladores\IProyectoCompiladoresEInterpretes\programa\OLIACCO
MPILER\target> java -jar OLIAC-1.0-SNAPSHOT-jar-with-dependencies.jar -f "ejemplo2.c"

>Error semantico en la linea 15: El valor retornado float no corresponde con el tipo de retorno string de la funcion determinarEstado.

--- Análisis semántico finalizado ---

Cantidad de errores semánticos detectados: 1

En línea 15 se observa que se está devolviendo un flotante, pero la función devuelve un string, eso da un problema semántico.

Generación de código intermedio:

Prueba 1

```

float obtenerNotaFinal( int nota1, int nota2, int nota3 ) {
    int suma = nota1 + nota2 + nota3;
    float promedio = suma / 3;
    return promedio;
}

string determinarEstado( float notaFinal ) {
    if(notaFinal >= 90) {
        return "Excelente";
    }
    elif(notaFinal >= 70) {
        return "Aprobado";
    }
    else {
        return "Reprobado";
    }
}

string mostrarEstudiante( string nombre, float notaFinal ) {
    writeString -> "Nombre del estudiante:"?
    writeString -> nombre?
    writeString -> "Nota final:"?
    writeFloat -> notaFinal?
    string estado = determinarEstado(notaFinal)?
    writeString -> "Estado:"?
    writeString -> estado?
}

void main() {
    int cantidadEstudiantes = 2;
    int i = 0;
    int nombres[2] = {2, 3};
    int notasFinales[2] = {5, 1000000000};

    for(i = 0; i < cantidadEstudiantes; ++i) {
        writeString -> "Ingrese las 3 notas del estudiante:"?
        int n1 = 0;
        int n2 = 0;
        int n3 = 0;
        readInt <- n1;
        readInt <- n2;
        readInt <- n3;
        float nf = obtenerNotaFinal(n1, n2, n3);
        notasFinales[i] = nf;
    }

    writeString -> "Mostrando resultados:"?
    for(i = 0; i < cantidadEstudiantes; ++i) {
        string nombre = "nombres[i]"?
        float nota = 7.056845105;
        mostrarEstudiante(nombre, nota);
    }

    writeString -> "Fin del programa."?
}

```

```

funcion obtenerNotaFinal:
param int nota1
param int nota2
param int nota3
t0 = nota1 + nota2
t1 = t0 + nota3
int suma = t1
t2 = suma / 3
float promedio = t2
return promedio
funcion determinarEstado:
param float notaFinal
et_if_else_inicio_0:
t3 = notaFinal >= 90
et_if_inicio_3:
if t3 goto et_if_intermedio_4
goto et_if_final_5
et_if_intermedio_4:
return "Excelente"
goto et_if_else_final_2
et_if_final_5:
t4 = notaFinal >= 70
et_if_inicio_6:
if t4 goto et_if_intermedio_7
goto et_if_final_8
et_if_intermedio_7:
return "Aprobado"
goto et_if_else_final_2
et_if_final_8:
return "Reprobado"
goto et_if_else_final_2
et_if_else_final_2:
funcion mostrarEstudiante:
param string nombre
param float notaFinal
t5 = "Nombre del estudiante:" WRITE_STRING t5
t6 = nombre WRITE_STRING t6
t7 = "Nota final:" WRITE_STRING t7
t8 = notaFinal WRITE_FLOAT t8
call = determinarEstado
string estado = determinarEstado()
t9 = "Estado:" WRITE_STRING t9
t10 = estado WRITE_STRING t10
funcion main:
int cantidadEstudiantes = 2
int i = 0
t11 = {2,3}
intArray nombres = t11
t12 = {5,1000000000}
intArray notasFinales = t12
et_for_inicio_9:
i = 0
et_for_intermedio_10:
t13 = i < cantidadEstudiantes
iffalse t13 goto et_for_final_11
t14 = i + 1
i = t14
t15 = "Ingrese las 3 notas del estudiante:" WRITE_STRING t15

```

El código intermedio se genera sin ningún problema.

Descripción del problema.

El problema central radica en diseñar e implementar un compilador que sea capaz de interpretar correctamente los elementos sintácticos y semánticos de este lenguaje, asegurando

que las instrucciones del usuario se traduzcan en un código intermedio correctamente estructurado y ejecutable en una etapa posterior. Para lograr esto, es indispensable abordar múltiples retos: desde la construcción de un analizador léxico que identifique los tokens del lenguaje, hasta la implementación de un analizador sintáctico capaz de validar la estructura del programa, pasando por el análisis semántico que verifica tipos, declaraciones y uso correcto de variables y funciones.

Además, es necesario garantizar una buena gestión de errores para proporcionar retroalimentación útil al usuario, así como generar un código intermedio que sirva de base para una futura etapa de traducción a código máquina o interpretación directa. Este código debe representar con precisión la lógica del programa, considerando instrucciones de control como condicionales, ciclos, llamadas a funciones y operaciones aritméticas o lógicas.

Por lo tanto, el problema que se busca resolver es la construcción de un compilador completo que procese programas escritos en un lenguaje propio, asegurando su validez léxica, sintáctica y semántica, y generando un código intermedio que represente de forma clara y ejecutable la lógica del programa fuente.

Diseño del programa: decisiones de diseño, algoritmos usados.

1. Análisis léxico

- **Herramienta utilizada:** JFlex.
- **Archivo:** Lexer.flex.
- **Decisiones de diseño:**
 - Se definieron patrones específicos para los tokens del lenguaje, incluyendo palabras reservadas (if, else, while, return, main, etc.), operadores, literales y comentarios.
 - Se incluyen comentarios de línea y bloque como tokens válidos para ser ignorados por el parser.
 - Se usan expresiones regulares con acciones asociadas para retornar objetos Symbol que contienen el tipo y valor del token.

2. Análisis sintáctico

- **Herramienta utilizada:** Java CUP.

- **Archivo:** Parser.cup.
- **Decisiones de diseño:**
 - Se definió una gramática LALR(1) modular con no terminales para sentencias (stmt), expresiones (expr), declaraciones (var_decl, func_decl), bloques (block) y estructuras de control (if, while, for, etc.).
 - La gramática incluye soporte completo para funciones con parámetros, return, llamadas y verificación semántica de tipo y cantidad de argumentos.
 - Se utiliza una pila de símbolos (CUP\$parser\$stack) para obtener referencias directas a tokens en caso de errores semánticos.

3. Análisis semántico

- **Estructura de símbolos:**
 - Se implementa una tabla de símbolos por función usando la clase TablaDeSimbolos, con soporte jerárquico para bloques anidados (escopos).
 - Cada variable o parámetro se representa como una instancia de la clase LineaTabla, que almacena su tipo, nombre, posición en el código y si ha sido inicializado.
 - Las funciones están encapsuladas en la clase Function, que incluye el tipo de retorno, nombre, argumentos y tabla de símbolos asociada.
- **Validaciones semánticas:**
 - Tipos compatibles en operaciones aritméticas, lógicas y relacionales.
 - Declaración y uso previo de identificadores.
 - Tipado fuerte y explícito en asignaciones, retornos de funciones y comparaciones.
 - Detección de duplicidad en nombres de variables y funciones.

4. Generación de código intermedio

- **Estructura usada:** InstrucionIntermedia y sus subclases como AsignacionInstr, OperacionInstr, EtiquetaEstructura, etc.
- **Decisiones de diseño:**
 - Cada instrucción relevante (declaración, asignación, operación, salto condicional) se traduce a una clase intermedia y se almacena en una lista global codigoIntermedio.
 - Se utiliza un contador de temporales (t0, t1, ...) y etiquetas (et_if_0, et_else_1, ...) para representar expresiones y estructuras de control.
 - El código generado se exporta a un archivo de texto mediante el método escribirCodigoIntermedio().

Librerías usadas:

- **picocli**: Esta es la librería que permite al usuario ingresar parámetros de manera más intuitiva.
- **Java cup**: permite realizar el parser de los tokens ingresados.
- **Java flex**: permite convertir los lexemas ingresados en el código fuente a tokens.
- **Java.util.ArrayList**: Permite almacenar dinámicamente las instrucciones intermedias generadas durante el análisis semántico del programa.
- **Java.io.File**: Representa archivos físicos que se van a generar para el almacenamiento del código intermedio.
- **Java.io.FileWriter**: Permite escribir secuencialmente líneas de texto en archivos, lo cual es esencial para exportar el código intermedio generado.
- **Java.util.HashMap**: Almacena las funciones definidas por el usuario y permite acceder a ellas por nombre de forma eficiente.

Análisis de resultados.

Objetivo	Estado	Descripción / Comentario
Implementar análisis semántico con tipado fuerte y explícito	<input checked="" type="checkbox"/> Cumplido	Se valida la compatibilidad de tipos en asignaciones, operaciones y retornos de funciones.
Detectar y reportar errores semánticos	<input checked="" type="checkbox"/> Cumplido	Se generan mensajes detallados de errores semánticos, incluyendo uso de variables no declaradas o mal tipadas.
Generar código intermedio de tres direcciones	<input checked="" type="checkbox"/> Cumplido	Se implementa generación de código intermedio usando instrucciones AsignacionInstr, OperacionInstr, etc.
Escribir el código intermedio en un archivo	<input checked="" type="checkbox"/> Cumplido	Se genera un archivo externo .txt con el código intermedio completo del programa fuente.
Uso de técnicas de recuperación en modo pánico	<input checked="" type="checkbox"/> Cumplido	Se utiliza recuperación de errores para seguir el análisis

		tras encontrar errores sintácticos o semánticos.
Compatibilidad con JFlex y CUP	<input checked="" type="checkbox"/> Cumplido	El analizador léxico y sintáctico fue generado correctamente con ambas herramientas.
Manejo de arreglos y matrices	No se cumple, no se generan errores semánticos ni se escribe en código intermedio	

Bitácora (autogenerada en git, commit por usuario incluyendo comentario).

<https://github.com/AndrewLopezHerrera/ProyectoCompiladoresEInterpretes/tree/main>