COMPILADORES E INTERPRETES

PROYECTO 3

ESTUDIANTES:

ESTEFANI VALVERDE

KEINGELL MOODIE

PROFESOR

ALLAN RODRIGUEZ

I SEMESTRE 2025

Manual de usuario: instrucciones de compilación, ejecución y uso bien detalladas.

Requisitos previos

Antes de compilar el proyecto, asegúrese de tener instalado:

• Java Development Kit (JDK) 17 o superior.

• Gradle 7.5 o superior.

• Las siguientes librerías:

• JFlex (generador de analizadores léxicos para Java).

• CUP (Constructor of Useful Parsers, generador de analizadores sintácticos).

• Java CUP Runtime (para ejecutar el parser generado por CUP)

- QtSpim 9.1 instalado

A la hora de crear el proyecto asegúrese de que el proyecto contenga un archivo build.gradle y

la estructura estándar:

/src

/main

/java

build.gradl

• Compilar el proyecto

• Ejecute el siguiente comando:

gradle build

• Ejecutar el programa

• Una vez compilado, ejecute el proyecto con:

gradle run

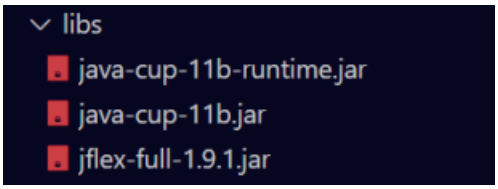
(Opcional) Limpiar archivos compilados

• Si desea limpiar los archivos generados por la compilación:

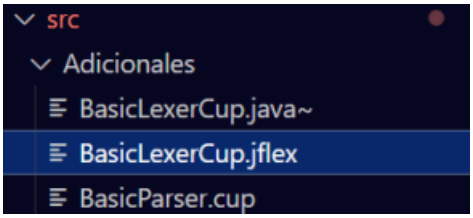
gradle clean

Pruebas de funcionalidad: Incluir screenshots.

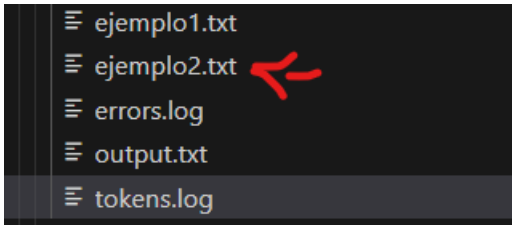
1 Como primer paso se debe tener incluidas las librerías de Jflex y Cup en la carpeta libs.



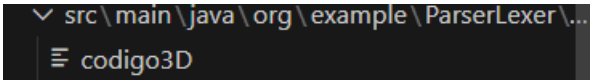
2 Como segundo paso se debe tener incluida la gramática elaborada en la asignación de la tarea 1 en los archivos .jflex y .cup para generar los archivos creados por el analizador léxico y sintáctico.



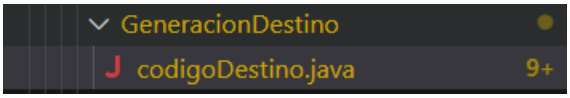
3 Un archivo de prueba que se utilizará para poner en practica el lenguaje creado con la gramática y hacer sus análisis léxicos, sintácticos, semánticos, como su código de 3 direcciones y código destino



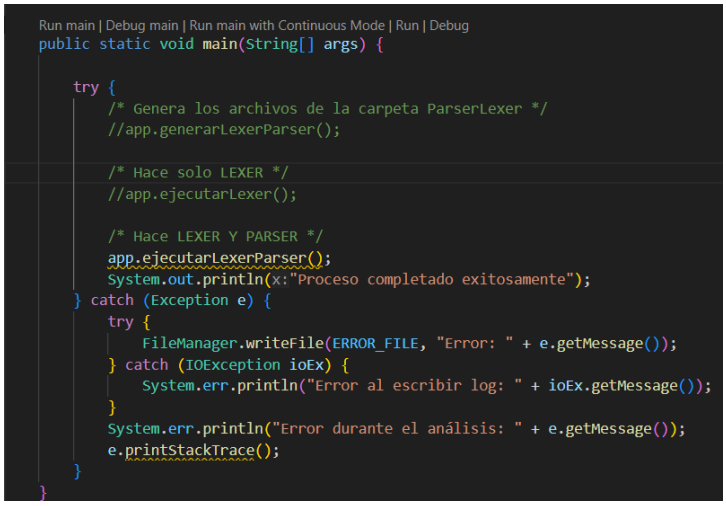
4 Ya que este proyecto es el final, también es necesario incluir el archivo de Codigo de tres direcciones creado a base de nuestro lenguaje.



5 En esta etapa final a base del código de tres direcciones tenemos que generar nuestro camino a nuestro código destino que será reflejado en el programa QtSpim, esto se validara en el siguiente archivo.

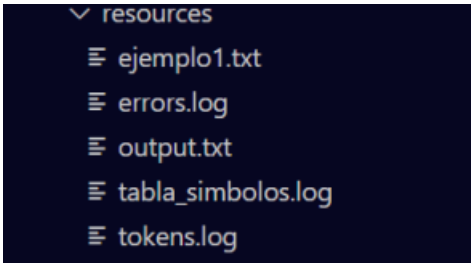


6 Ya con esto se puede pasar al momento de ejecución, ejecutando el programa en el archivo main que es el App.java



7 Inmediatamente el programa generara los archivos BasicLexerCup.java, parser.java y sym.java quienes son archivos generados por el Jflex y el Cup, donde se harán todos los análisis antes mencionados.

8 El programa en su tiempo de ejecución va generar los archivos de error, tabla de símbolos y los tokens.



9 Por consola también se van a reflejar la mayoría de los análisis en ejecución, se va reflejar su código de tres direcciones y su código destino con éxito.

```
Inicializando analizador lexico...
Analizador lexico inicializado correctamente.
ErrorHandler conectado desde lexer a parser
Iniciando análisis sintáctico...
Variable agregada: msg:string en scope: global
Entrando al scope: main
Variable agregada: var3:int en scope: main
Variable agregada: var4:int en scope: main
Entrando al scope: main_block_1
Entrando al scope: main_block_1_block_2
Saliendo del scope: main_block_1_block_2
Saliendo del scope: global
Variable agregada: i:int en scope: global
Temp ====================> t27Valor =======> int
Entrando al scope: global_block_3
Variable agregada: msg1:string en scope: global_block_3
Saliendo del scope: global_block_3
Variable agregada: var1:int en scope: global
Entrando al scope: global_block_4
Temp ====================> t34Valor =======> desconocido
Saliendo del scope: global_block_4
Variable agregada: x1:int en scope: global
Variable agregada: array:int[int] en scope: global
Temp ====================> t45Valor =======> int
Variable agregada: x:int en scope: global
Variable agregada: array1:int[int][int] en scope: global
Temp ====================> t54Valor =======> int
Saliendo del scope: global
Entrando al scope: f1
Variable agregada: num1:int en scope: f1
Variable agregada: num2:int en scope: f1
```

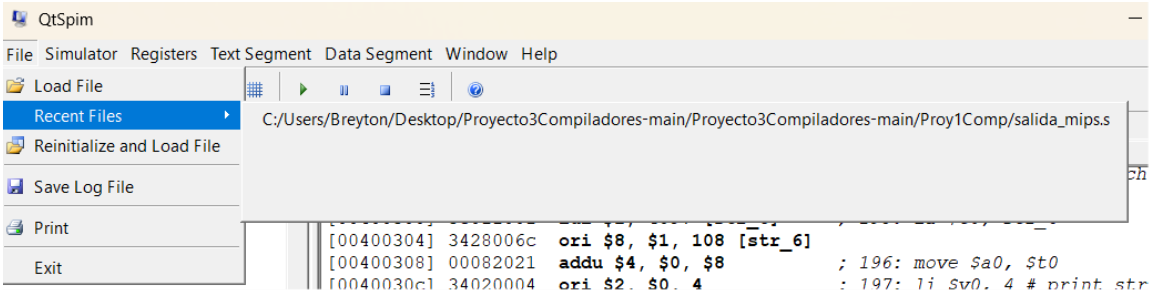
```
=== TABLA DE SÍMBOLOS ===
Scope: main_block_1
Scope: global_block_3
  msg1:string
Scope: global_block_4
Scope: global
  msg:string
  i:int
  var1:int
  x1:int
  array:int[int]
  x:int
  array1:int[int][int]
  f1:function(int_num1,int_num2)->int
  f2:function(int_num1,string_num2)->string
Scope: main
  var3:int
  var4:int
Scope: f1
  num1:int
  num2:int
Scope: f2
  num1:int
  num2:string
Scope: main_block_1_block_2
========================
```

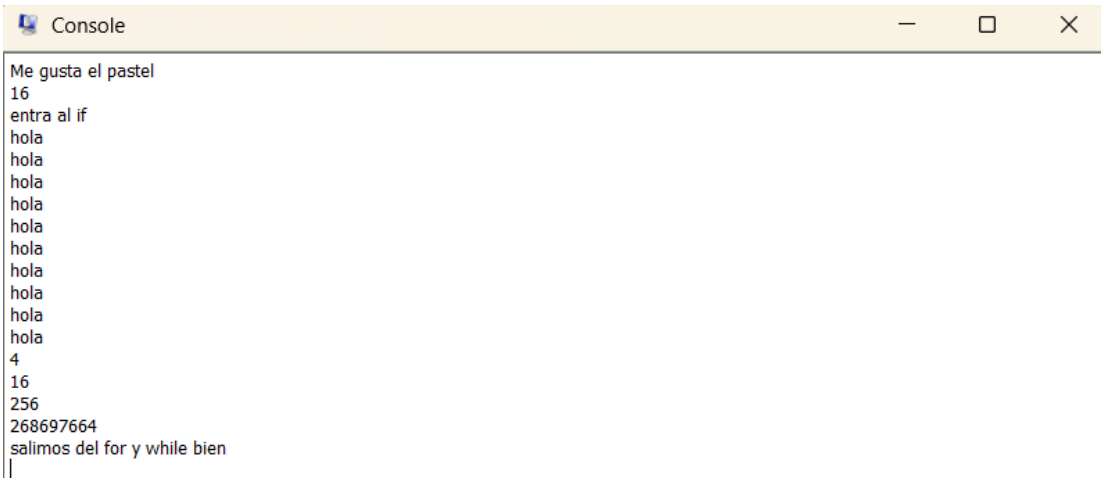
```
=== CÓDIGO INTERMEDIO ===

t1 = "hola";
declaracion_global_1: msg = t1;
INICIO_main:
t2 = 2;
t3 = "Me gusta el pastel";
llamada_1: t4 = call f2([t2, t3]);
t5 = 2;
t6 = 4;
llamada_2: t7 = call f1([t5, t6]);
t8 = 5;
declaracion_2: var3 = t8;
t9 = 6;
declaracion_3: var4 = t9;
t10 = var3;
t11 = 5;
t12 = t10 == t11;
t13 = var4;
t14 = 6;
t15 = t13 == t14;
t16 = t12 && t15;
INICIO_if_1:
if (t16) goto if_1_true;
goto FIN_if_1_bloque;
```

```
Código intermedio guardado en: C:\Users\
le\ParserLexer\resources\codigo3D
Liberado registro $t0 para temporal t1
Liberado registro $t1 para temporal t2
Liberado registro $t2 para temporal t3
Liberado registro $t3 para temporal t4
Liberado registro $t4 para temporal t5
Liberado registro $t5 para temporal t6
Liberado registro $t6 para temporal t7
Liberado registro $t7 para temporal t8
Liberado registro $t8 para temporal t9
Liberado registro $t9 para temporal t10
Liberado registro $t0 para temporal t11
Liberado registro $t3 para temporal t14
Liberado registro $t2 para temporal t13
Liberado registro $t1 para temporal t12
Liberado registro $t4 para temporal t15
Liberado registro $t5 para temporal t16
  No se reconoce patrón para línea:
Liberado registro $t6 para temporal t17
  No se reconoce patrón para línea:
Liberado registro $t7 para temporal t19
  No se reconoce patrón para línea:
Liberado registro $t8 para temporal t21
Liberado registro $t0 para temporal t23
Liberado registro $t9 para temporal t22
```

10 En el programa QtSpim se verán los resultados del código intermedio por consola, recordar cargar el archivo y correrlo.





Descripción del problema

Incluyendo el proyecto 1 y proyecto 2 que se basan en los análisis léxico, sintáctico y semántico con sus correcciones, esta etapa 3 del proyecto consiste en la generación de código destino, debe utilizarse el código 3D generado por el CUP durante el análisis semántico para, basado en el, generar código MIPS.

Diseño del programa

El programa desarrollado implementa un compilador básico para un lenguaje de alto nivel personalizado con sintaxis simplificada (tipo pseudocódigo) que genera código intermedio (3D) y lo traduce a código ensamblador MIPS. El diseño se centra en modularidad, legibilidad, y una transición clara entre las fases del compilador: análisis, traducción y ejecución.

Decisiones de Diseño

1. **Separación por fases:**
    - Separamos claramente las etapas de análisis léxico/sintáctico, generación de código intermedio (3D), y traducción a MIPS para facilitar el mantenimiento y escalabilidad.
2. **Representación de código intermedio:**
    - Utilizamos una sintaxis de tres direcciones (3D) para representar instrucciones en forma tX = operación, lo que facilita su traducción directa a registros en MIPS.
    - Esta representación es clave para manejar expresiones, llamadas, asignaciones, arreglos y estructuras de control de forma uniforme.
3. **Asignación de registros:**
    - Se implementó un sistema dinámico de asignación de registros temporales ($t0-$t7, $f0-$f31), con un mecanismo de asignación y liberación mediante mapas (HashMap).
    - Esto permite reutilizar registros de forma eficiente sin colisiones entre llamadas a funciones o ciclos.
4. **Traducción a MIPS:**
    - Cada patrón de 3D tiene su correspondiente función de traducción: operaciones aritméticas, asignaciones, llamadas a funciones, arreglos, ciclos for, condicionales, etc.
    - El uso de expresiones regulares (Regex) facilita la detección de patrones y la organización del traductor.
5. **Soporte para strings y arreglos:**
    - Los strings se manejan como punteros a secciones .asciiz en el segmento .data.
    - Los arreglos se simulan con .space en .data para tamaños fijos y se acceden mediante cálculo de offsets con mul y add.

6. **Ejecución desde función main:**
    - Toda ejecución comienza desde la etiqueta main: y se invocan funciones auxiliares con preservación del registro de retorno ($ra) mediante el stack ($sp).

Algoritmos usados

1. **Asignación dinámica de registros**:
   Se implementa un algoritmo basado en disponibilidad para asignar registros enteros o flotantes según el tipo del valor temporal. Cuando se libera una variable temporal, su registro se reincorpora a la lista de disponibles.
2. **Traducción de arreglos**:
   Para simular acceso a arreglos, se calcula el offset como offset = índice * 4, luego se accede a la dirección base + offset para almacenar o recuperar datos.
3. **Evaluación de expresiones aritméticas**:
   Las operaciones se traducen directamente en instrucciones como add, sub, mul, y para potencia se simula mediante un loop multiplicativo.
4. **Manejo de ciclos for**:
   Se traduce usando etiquetas INICIO_for, for_true, y FIN_for, junto con saltos condicionales (bne, j) y evaluación previa de la condición y el incremento.
5. **Traducción de llamadas a funciones**:
   Los parámetros se pasan en registros $a0-$a3, se hace jal, y el resultado se captura en $v0. El contexto del caller se preserva con sw $ra en el stack.
6. **Control de flujo**:
   Las estructuras condicionales y bucles usan instrucciones beqz, bne, y saltos etiquetados para simular el flujo del código original.

Librerías usadas:

Para la correcta elaboración del proyecto se utilizaron las siguientes librerías y herramientas:

- Gradle (automatización de construcción de proyectos Java).
- Java Development Kit (JDK)
- JFlex (generador de analizadores léxicos para Java).
- CUP (Constructor of Useful Parsers, generador de analizadores sintácticos).
- Java CUP Runtime (para ejecutar el parser generado por CUP)

Análisis de resultados: objetivos alcanzados, objetivos no alcanzados.

Objetivo	Alcanzado	No Alcanzado
El sistema debe leer un archivo fuente.	✓	
Se debe escribir en un archivo todos los tokens encontrados, identificador asociado con el lexema.	✓	
Por cada token deberán indicar en cuál tabla de símbolos va y cual información se almacenará.	✓	
Indicar si el archivo fuente puede o no ser generado por la gramática.	✓	
Reportar y manejar los errores léxicos y sintácticos encontrados.	✓	
Generación de Código intermedio	✓	
Generación de Código Destino	✓	

Bitácora (autogenerada en git)

https://github.com/Estefani05/Proyecto3Compiladores