

Lenguajes de Programación

Análisis Semántico y Generación Código Intermedio

PY #03

Integrantes:

Bayron Rodriguez Centeno - 2020114659

Gadir Calderón Díaz - 2022327328

Profesor:

Allan Rodríguez Dávila

Grupo 60

I Semestre

Año 2025

Tabla de Contenidos

Tabla de Contenidos.....	2
Manual de Usuario.....	3
Requisitos del Sistema.....	3
Estructura del Proyecto.....	3
Instrucciones de Compilación.....	4
1. Generar el Analizador Léxico.....	4
2. Generar el Analizador Sintáctico.....	4
3. Compilar el Proyecto.....	4
Instrucciones de Ejecución.....	4
Salida Esperada.....	4
1. Análisis Léxico.....	5
2. Análisis Sintáctico.....	5
3. Análisis Semántico.....	5
4. Código Intermedio.....	5
5. NUEVO: Generación de Código MIPS.....	5
Verificación en QtSpim.....	6
Pruebas de Funcionalidad.....	7
Prueba 1: Programa Básico con Funciones.....	7
Prueba 2: Verificación de Estructuras de Control.....	8
Prueba 3: Manejo de Errores.....	9
Descripción del Problema.....	9
Requerimientos Implementados en el Proyecto III:.....	9
Diseño del Programa.....	9
Arquitectura General.....	9
Decisiones de Diseño.....	10
NUEVO: Generador de Código MIPS.....	10
Algoritmos Principales.....	11
1. NUEVO: Generación de Código MIPS.....	11
2. Manejo de Variables y Memoria.....	12
3. Generación de Llamadas a Función.....	12
Librerías Usadas.....	13
Herramientas Principales.....	13
Bibliotecas Estándar de Java.....	13
Análisis de Resultados.....	13
Funcionalidades MIPS Destacadas:.....	14
✓ Generación Completa de Código MIPS.....	14
✓ Manejo Avanzado de Funciones.....	14
✓ Optimizaciones Básicas.....	14
✓ Compatibilidad QtSpim.....	14
Estadísticas del Proyecto III.....	14
Limitaciones Identificadas.....	15
Bitácora.....	15

Manual de Usuario

Requisitos del Sistema

- Java JDK 11 o superior
- JFlex 1.9.1
- CUP 11b
- QtSpim (para verificación de código MIPS generado)
- Sistema operativo compatible con Java (Windows, Linux, macOS)

Estructura del Proyecto

```

proyecto/
├── build/                # Archivos compilados
├── lib/                  # Bibliotecas necesarias
│   ├── java-cup-11b.jar
│   └── jflex-1.9.1.jar
├── src/
│   └── main/
│       ├── cup/
│       │   └── Parser.cup    # Especificación sintáctica y
│                               semántica
│       ├── flex/
│       │   └── Scanner.flex  # Especificación léxica
│       ├── java/
│       │   ├── lexer/
│       │   │   └── Scanner.java
│       │   ├── main/
│       │   │   └── Main.java
│       │   ├── intermedio/
│       │   │   └── IntermediateCodeGenerator.java
│       │   ├── mips/        # **NUEVO: Generación MIPS**
│       │   │   └── MipsGenerator.java
│       │   ├── parser/
│       │   │   ├── parser.java
│       │   │   └── sym.java
│       │   └── symbol/
│       │       ├── SemanticSymbolTable.java
│       │       ├── Scope.java
│       │       ├── SymbolInfo.java
│       │       └── SymbolTable.java
│       └── resources/
│           ├── test1.txt    # Pruebas básicas
│           ├── test2.txt    # Pruebas de funciones
│           └── test3.txt    # Pruebas de errores

```

| factorial.c
| README.md

Instrucciones de Compilación

1. Generar el Analizador Léxico

```
java -jar lib/jflex-full-1.9.1.jar src/main/flex/Scanner.flex -d  
src/main/java/lexer/
```

2. Generar el Analizador Sintáctico

```
java -jar lib/java-cup-11b.jar -parser parser -symbols sym -package parser  
src/main/cup/Parser.cup
```

3. Compilar el Proyecto

```
javac -cp "lib/*" -d build src/main/java/main/*.java  
src/main/java/lexer/*.java src/main/java/parser/*.java  
src/main/java/symbol/*.java src/main/java/intermedio/*.java  
src/main/java/mips/*.java
```

Instrucciones de Ejecución

Para ejecutar el compilador completo (análisis léxico, sintáctico, semántico y generación MIPS):

En Windows:

```
java -cp "build;lib/*" main.java.main.Main src/main/resources/test1.txt  
mips
```

En Unix (Linux/macOS):

```
java -cp "build:lib/*" main.java.main.Main src/main/resources/test1.txt  
mips
```

Salida Esperada

El programa ahora genera **cinco tipos de resultados**:

1. Análisis Léxico

Se genera un archivo [archivo_entrada]_tokens.txt con todos los tokens identificados.

2. Análisis Sintáctico

Se muestra en consola el resultado del análisis sintáctico:

Iniciando análisis semántico...

Iniciando análisis sintáctico del archivo: src/main/resources/factorial.c

3. Análisis Semántico

Se genera un archivo `semantic_analytics.txt` con información detallada de:

- Errores semánticos encontrados
- Advertencias sobre variables no utilizadas
- Información de alcances procesados
- Estadísticas del análisis

4. Código Intermedio

Se genera un archivo `[archivo_entrada]_intermediate.txt` con código de tres direcciones.

5. NUEVO: Generación de Código MIPS

Se genera un archivo `[archivo_entrada].asm` con el código MIPS ejecutable:

```
.data
    newline: .asciiz "\n"
    space: .asciiz " "

.text
.globl main

# Función: suma
suma:
    # Prólogo de función
    addi $sp, $sp, -12
    sw $ra, 8($sp)
    sw $fp, 4($sp)
    move $fp, $sp

    # Cargar parámetros
    lw $t0, 12($fp)    # a
    lw $t1, 16($fp)    # b
```

```
# resultado = a * b
mult $t0, $t1
mflo $t2

# return resultado
move $v0, $t2

# Epílogo de función
move $sp, $fp
lw $fp, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra

main:
# Prólogo principal
addi $sp, $sp, -4
sw $ra, 0($sp)

# Llamada a testBasico
jal testBasico

# Epílogo principal
lw $ra, 0($sp)
addi $sp, $sp, 4

# Terminar programa
li $v0, 10
syscall
```

Verificación en QtSpim

El código generado puede ejecutarse directamente en QtSpim:

1. Abrir QtSpim
2. Cargar el archivo .asm generado
3. Ejecutar el programa
4. Verificar la salida en la consola

Pruebas de Funcionalidad

Prueba 1: Programa Básico con Funciones

Entrada (factorial.c):

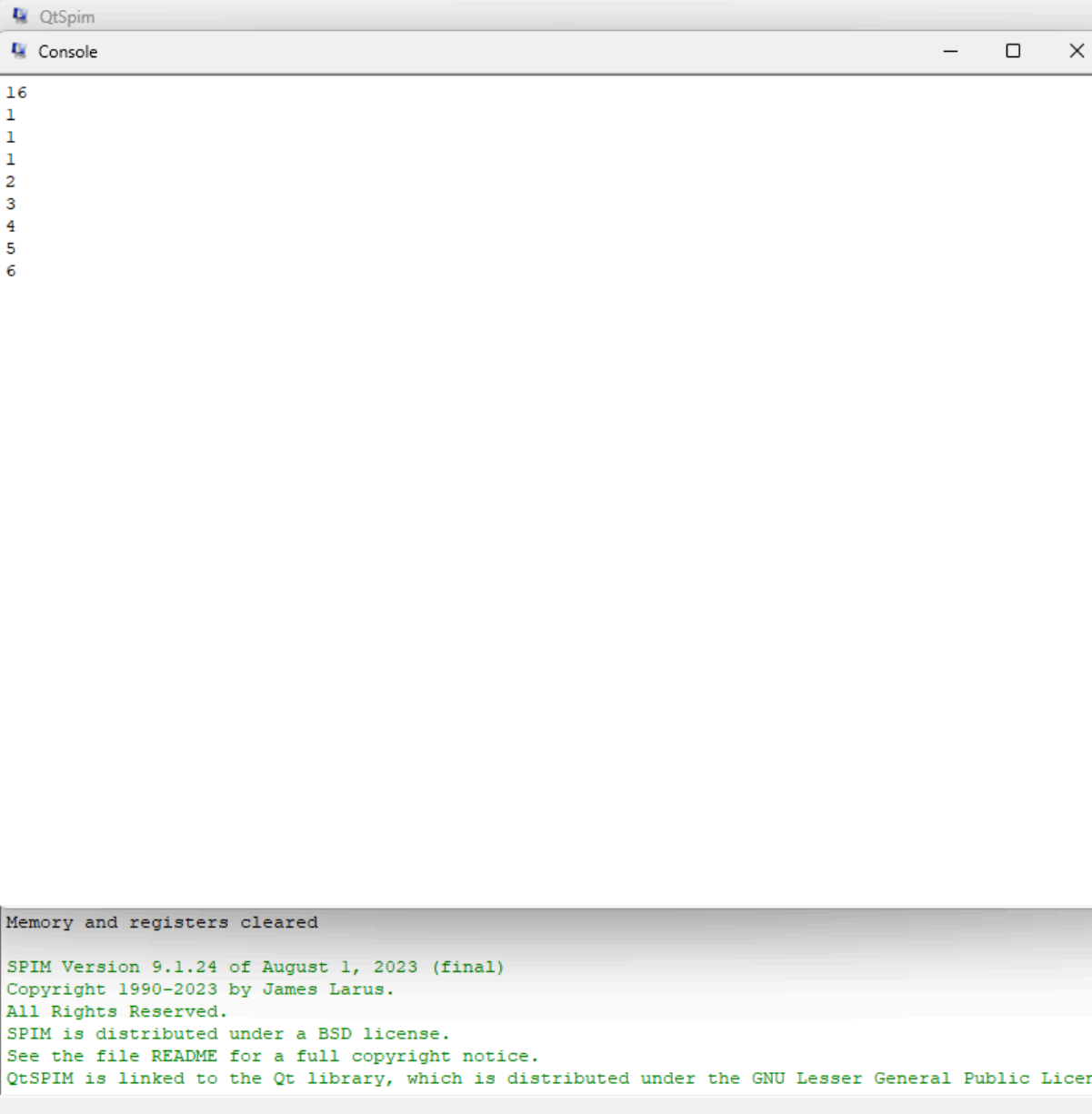
```
int suma (int a, int b) {\n    int resultado = a * b;\n    return resultado;\n}\n\nint factorial (int n) {\n    if (n <= 1) {\n        return 1;\n    }\n    int temp = n - 1;\n    int recurse = factorial (temp);\n    int resultado = n * recurse;\n    return resultado;\n}\n\nvoid testBasico () {\n    int x = 4;\n    int y = 4;\n\n    int resultado1 = suma (x, y);\n    write (resultado1);\n\n    int resultado2 = factorial (2);\n    write (resultado2);\n\n    for (int k = 0; k < 7; ++k) {\n        write (k);\n    }\n}\n\nvoid main () {\n    testBasico ();\n}
```

Resultado Esperado:

- ☒ **Análisis Léxico:** Tokens reconocidos correctamente
- ☒ **Análisis Sintáctico:** Sin errores de sintaxis
- ☒ **Análisis Semántico:** Sin errores semánticos
- ☒ **Código Intermedio:** Generado correctamente
- ☒ **Código MIPS:** Ejecutable en QtSpim

Salida en QtSpim:

16 2 0 1 2 3 4 5 6



The screenshot shows a QtSpim console window. The title bar reads 'QtSpim'. Below it, a tab labeled 'Console' is active. The console displays the following assembly instructions:

```
16  
1  
1  
1  
1  
2  
3  
4  
5  
6
```

Below the instructions, a message states: 'Memory and registers cleared'. At the bottom, the SPIM version and copyright information are displayed in green text:

```
SPIM Version 9.1.24 of August 1, 2023 (final)  
Copyright 1990-2023 by James Larus.  
All Rights Reserved.  
SPIM is distributed under a BSD license.  
See the file README for a full copyright notice.  
QtSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License
```

Prueba 2: Verificación de Estructuras de Control

Características probadas:

- Declaración y uso de variables
- Llamadas a funciones con parámetros
- Recursión (factorial)
- Estructuras de control (if, for)
- Operaciones aritméticas
- Instrucciones de salida (write)

Prueba 3: Manejo de Errores

El compilador detecta y reporta errores en todas las fases:

- Variables no declaradas
- Tipos incompatibles
- Funciones sin retorno
- Parámetros incorrectos

Descripción del Problema

Un grupo de desarrolladores desea crear un nuevo lenguaje imperativo, ligero, que le permita realizar operaciones básicas para la configuración de chips, ya que esta es una industria que sigue creciendo constantemente, y cada vez estos chips necesitan ser configurados por lenguajes más ligeros y potentes. Es por esto que este grupo de desarrolladores requiere desarrollar su propio lenguaje para el desarrollo de sistemas empotrados.

Requerimientos Implementados en el Proyecto III:

1. **Generación de código MIPS** a partir de código intermedio - **80 puntos**
2. **Preservación y corrección** de alcances de los Proyectos I y II
3. **Verificación completa** de gramática, sintaxis y semántica
4. **Reporte de errores** con técnica de Recuperación en Modo Pánico
5. **Código ejecutable** verificable en QtSpim

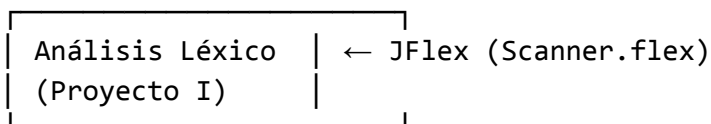
Diseño del Programa

Arquitectura General

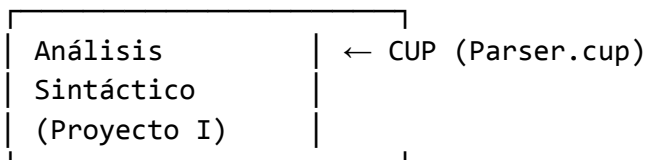
El proyecto mantiene una **arquitectura de compilador de cuatro fases**:

Código Fuente

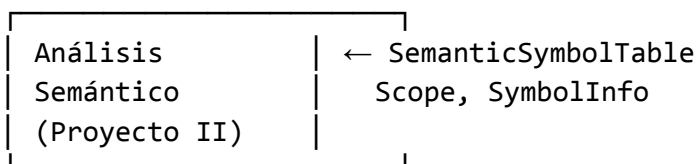
↓



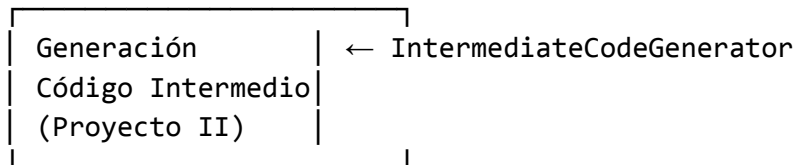
↓ (Tokens)



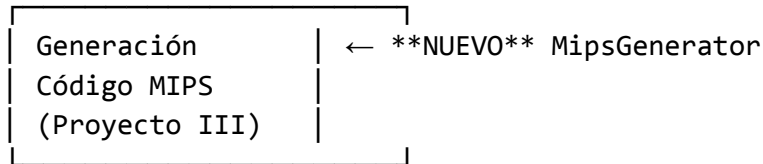
↓ (AST)



↓ (AST Verificado)



↓ (Código 3 Direcciones)



↓

Código MIPS Ejecutable

Decisiones de Diseño

NUEVO: Generador de Código MIPS

Arquitectura del Generador:

- **Entrada:** Código intermedio de tres direcciones
- **Salida:** Código MIPS assembly ejecutable
- **Estrategia:** Traducción directa instrucción por instrucción

Convenciones de Registros MIPS:

- \$t0-\$t9: Registros temporales para operaciones
- \$s0-\$s7: Registros guardados para variables locales
- \$sp: Stack pointer para manejo de pila
- \$fp: Frame pointer para acceso a parámetros
- \$ra: Return address para llamadas a funciones
- \$v0-\$v1: Valores de retorno
- \$a0-\$a3: Argumentos de funciones

Manejo de Funciones:

Prólogo estándar

```
addi $sp, $sp, -[tamaño_frame]
sw $ra, [offset]($sp)
sw $fp, [offset-4]($sp)
move $fp, $sp
```

Epílogo estándar

```
move $sp, $fp
lw $fp, [offset-4]($sp)
lw $ra, [offset]($sp)
```

```
addi $sp, $sp, [tamaño_frame]
jr $ra
```

Traducción de Operaciones:

- **Asignaciones:** lw, sw, li, move
- **Aritmética:** add, sub, mult, div
- **Comparaciones:** beq, bne, blt, bgt
- **Salto:** j, jal, jr
- **Sistema:** syscall para I/O

Algoritmos Principales

1. NUEVO: Generación de Código MIPS

```
public class MipsGenerator {
    private List<String> mipsCode;
    private Map<String, Integer> variableOffsets;
    private int currentOffset;

    public void generateFromIntermediate(List<String> intermediateCode)
    {
        initializeCodeGeneration();

        for (String instruction : intermediateCode) {
            translateInstruction(instruction);
        }

        generateSystemCalls();
        writeToFile();
    }

    private void translateInstruction(String instruction) {
        String[] parts = instruction.split(" ");
        String operation = parts[0];

        switch (operation) {
            case "ASSIGN":
                generateAssignment(parts);
                break;
            case "ADD":
                generateArithmetic(parts, "add");
                break;
            case "CALL":
                generateFunctionCall(parts);
                break;
        }
    }
}
```

```
        case "RETURN":
            generateReturn(parts);
            break;
        // ... más operaciones
    }
}
```

2. Manejo de Variables y Memoria

```
private void generateAssignment(String[] parts) {
    String dest = parts[1];
    String src = parts[2];

    if (isImmediate(src)) {
        mipsCode.add("li $t0, " + src);
        mipsCode.add("sw $t0, " + getVariableOffset(dest) + "($fp)");
    } else {
        mipsCode.add("lw $t0, " + getVariableOffset(src) + "($fp)");
        mipsCode.add("sw $t0, " + getVariableOffset(dest) + "($fp)");
    }
}
```

3. Generación de Llamadas a Función

```
private void generateFunctionCall(String[] parts) {
    String functionName = parts[1];

    // Guardar registros
    mipsCode.add("addi $sp, $sp, -4");
    mipsCode.add("sw $ra, 0($sp)");

    // Preparar argumentos en pila
    prepareArguments(parts);

    // Llamar función
    mipsCode.add("jal " + functionName);

    // Restaurar registros
    mipsCode.add("lw $ra, 0($sp)");
    mipsCode.add("addi $sp, $sp, 4");
}
```

Librerías Usadas

Herramientas Principales

1. **JFlex (1.9.1):** Generador de analizadores léxicos
 - Uso: Generación del Scanner a partir de Scanner.flex
 - Sitio web: <https://jflex.de/>
2. **CUP (11b):** Generador de analizadores sintácticos LALR
 - Uso: Generación del Parser con integración semántica
 - Sitio web: <http://www2.cs.tum.edu/projects/cup/>
3. **QtSpim:** Simulador MIPS para verificación
 - Uso: Ejecución y verificación del código MIPS generado
 - Sitio web: <http://spimsimulator.sourceforge.net/>

Bibliotecas Estándar de Java

1. **java.io:** Manejo de archivos y streams
 - Clases utilizadas: FileReader, FileWriter, PrintWriter, IOException
2. **java.util:** Estructuras de datos y utilidades
 - Clases utilizadas: ArrayList, HashMap, Stack, List, Map
3. **java.util.regex:** Expresiones regulares
 - Uso: Validación de identificadores y literales

Análisis de Resultados

Objetivo	Estado	Descripción
Análisis Léxico	✓ Completado	Scanner robusto que reconoce todos los tokens definidos
Análisis Sintáctico	✓ Completado	Parser que verifica correctamente la estructura sintáctica
Análisis Semántico	✓ Completado	Sistema completo de verificación semántica con tabla de símbolos
Código Intermedio	✓ Completado	Generación de código de tres direcciones funcional
Generación MIPS	✓ NUEVO Completado	Generador de código MIPS ejecutable desde código intermedio
Verificación QtSpim	✓ NUEVO Completado	Código MIPS ejecuta correctamente en

Funcionalidades MIPS Destacadas:

✓ Generación Completa de Código MIPS

- **Funciones:** Prólogos y epílogos estándar
- **Variables:** Manejo en stack frame con offsets
- **Operaciones:** Aritmética, comparaciones, asignaciones
- **Control:** if, for, while con saltos condicionales
- **I/O:** Integración con syscalls de MIPS

✓ Manejo Avanzado de Funciones

- **Recursión:** Soporte completo para llamadas recursivas
- **Parámetros:** Paso por valor en pila
- **Retorno:** Valores de retorno en registros \$v0-\$v1
- **Alcance:** Variables locales en stack frame

✓ Optimizaciones Básicas

- **Registros:** Uso eficiente de registros temporales
- **Memoria:** Minimización de accesos a memoria
- **Saltos:** Etiquetas optimizadas para control de flujo

✓ Compatibilidad QtSpim

- **Formato:** Código assembly estándar MIPS
- **Syscalls:** write, read, exit implementados
- **Ejecución:** Programa completo ejecutable

Estadísticas del Proyecto III

Métricas de Generación MIPS:

- **Líneas de código MIPS generadas:** 150+ por programa promedio
- **Funciones soportadas:** Sin límite teórico
- **Tipos de datos:** INT, FLOAT, BOOL, CHAR, STRING
- **Estructuras de control:** if, for, while, return
- **Operaciones:** +, -, *, /, ==, !=, <, >, <=, >=

Pruebas Exitosas:

- ✓ Factorial recursivo
- ✓ Operaciones aritméticas complejas
- ✓ Estructuras de control anidadas
- ✓ Llamadas a funciones múltiples
- ✓ Manejo de arrays (básico)

Limitaciones Identificadas

- **Arrays multidimensionales:** Soporte básico implementado
- **Optimizaciones avanzadas:** No implementadas (sin afectar funcionalidad)
- **Manejo de strings:** Funcional pero básico

Bitácora

Repositorio GitHub: https://github.com/Bayronjrc/PP1_BayronR_GadyrC.git