

Lenguajes de Programación

Análisis Semántico y Generación Código Intermedio

PY #02

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.....	2
Requisitos del Sistema.....	2
Estructura del Proyecto.....	2
Instrucciones de Compilación.....	3
1. Generar el Analizador Léxico.....	3
2. Generar el Analizador Sintáctico.....	3
3. Compilar el Proyecto.....	3
Instrucciones de Ejecución.....	3
Salida Esperada.....	3
1. Análisis Léxico.....	4
2. Análisis Sintáctico.....	4
3. NUEVO: Análisis Semántico.....	4
Formato del Archivo de Entrada.....	5
Pruebas de Funcionalidad.....	6
Prueba 1: Análisis Básico Sin Errores.....	6
Prueba 2: Verificación Semántica de Tipos.....	7
Descripción del Problema.....	9
Nuevos Requerimientos Implementados:.....	10
Diseño del Programa.....	10
Arquitectura General.....	10
Decisiones de Diseño.....	11
NUEVO: Analizador Semántico.....	11
Scanner y Parser (Mantenidos del Proyecto I).....	12
Main (Programa Principal).....	12
Algoritmos Principales.....	12
1. NUEVO: Análisis Semántico.....	12
2. Verificación de Tipos.....	13
3. Análisis Léxico (JFlex).....	14
4. Análisis Sintáctico (CUP).....	14
Librerías Usadas.....	14
Herramientas Principales.....	14
Bibliotecas Estándar de Java.....	14
Análisis de Resultados.....	14
Funcionalidades Semánticas Destacadas:.....	16
✓ Verificación de Tipos Estricta.....	16
✓ Manejo de Alcances Avanzado.....	16
✓ Verificación de Funciones Completa.....	16
✓ Arrays Bidimensionales.....	16
✓ Sistema de Errores Robusto.....	16
Estadísticas del Análisis Semántico.....	16
Puntos Extra No Implementados.....	16

Manual de Usuario

Requisitos del Sistema

- Java JDK 11 o superior
- JFlex 1.9.1
- Cub 11b
- 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
        ├── parser/
        │   ├── parser.java
        │   └── sym.java
        └── symbol/          # **NUEVO: Análisis Semántico**
            ├── SemanticSymbolTable.java
            ├── Scope.java
            ├── SymbolInfo.java
            └── SymbolTable.java
└── resources/
    ├── test1.txt           # Pruebas básicas
    ├── test2.txt           # Pruebas de funciones
    ├── test3.txt           # Pruebas de errores
    └── tata.c              # Archivo de prueba completo
  
```

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
```

Instrucciones de Ejecución

Para ejecutar el analizador completo (léxico, sintáctico y semántico):

En Windows:

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

En Unix (Linux/macOS):


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

Salida Esperada

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

1. Análisis Léxico

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

```
src > main > resources >  tata_tokens.txt
1 Token: FLOAT, Lexema: FLOAT, Línea: 3, Columna: 2, Tabla: PALABRAS_RESERVADAS
2 Token: ID, Lexema: func1, Línea: 3, Columna: 8, Tabla: VARIABLES
3 Token: LPAREN, Lexema: LPAREN, Línea: 3, Columna: 14, Tabla: N/A
4 Token: CHAR, Lexema: CHAR, Línea: 3, Columna: 16, Tabla: PALABRAS_RESERVADAS
5 Token: ID, Lexema: x22, Línea: 3, Columna: 21, Tabla: VARIABLES
6 Token: COMMA, Lexema: COMMA, Línea: 3, Columna: 24, Tabla: N/A
7 Token: CHAR, Lexema: CHAR, Línea: 3, Columna: 25, Tabla: PALABRAS_RESERVADAS
8 Token: ID, Lexema: x22, Línea: 3, Columna: 30, Tabla: VARIABLES
9 Token: RPAREN, Lexema: RPAREN, Línea: 3, Columna: 34, Tabla: N/A
10 Token: LBLOCK, Lexema: LBLOCK, Línea: 3, Columna: 37, Tabla: N/A
```

2. Análisis Sintáctico

Se muestra en consola el resultado del análisis sintáctico:

```
Iniciando analisis semantico...
Iniciando análisis sintáctico del archivo: src/main/resources/tata.c
```

3. NUEVO: Análisis Semántico

Se genera un archivo semantic_analytics.txt con información detallada:

```
semantic_analysis.txt
1  ===== ANÁLISIS SEMÁNTICO =====
2  Alcances procesados total: 6
3  Alcances activos: 1
4  Errores semánticos: 24
5  Advertencias: 13
6
7  ===== ERRORES SEMÁNTICOS =====
8  Variable 'x22' ya declarada en este alcance en línea 3
9  Variable 'x50' ya declarada en este alcance en línea 12
10 Variable 'var' no declarada en línea 13
11 Tipos incompatibles en asignacion: 'CHAR' no puede recibir 'INT' en línea 15
12 Variable 'x24' no declarada en línea 18
13 Variable 'var' no declarada en línea 18
14 Variable 'y' ya declarada en este alcance en línea 19
15 Tipos incompatibles en asignacion: 'CHAR' no puede recibir 'INT' en línea 20
16 Variable 'ch33' ya declarada en este alcance en línea 21
17 Variable 'y' ya declarada en este alcance en línea 24
18 Variable 'j' no declarada en línea 27
19 Función 'func1' de tipo 'FLOAT' debe retornar un valor en línea 31
20 Variable 'b1' usada sin inicializar en línea 35
21 Función 'func3' de tipo BOOL no puede retornar VARIABLE en línea 35
22 Función 'func2' de tipo 'BOOL' debe retornar un valor en línea 40
23 Variable 'fl1' ya declarada en este alcance en línea 50
24 Tipos incompatibles: esperado 'INT', encontrado 'NUMERIC' en línea 51
25 Variable 'fl1' usada sin inicializar en línea 52
26 Tipos incompatibles: esperado 'FLOAT', encontrado 'NUMERIC' en línea 52
27 Variable 'hola' no declarada en línea 55
28 Variable 'arr' no declarada en línea 55
29 Variable 'fl1' usada sin inicializar en línea 56
30 Variable 'fl1' usada sin inicializar en línea 60
31 Función 'main' de tipo VOID no puede retornar un valor en línea 61
32
33 ===== ADVERTENCIAS =====
34 Variable 'x50' declarada pero no utilizada en línea 11
35 Variable 'miChar2' declarada pero no utilizada en línea 8
36 Variable 'x30' declarada pero no utilizada en línea 9
37 Variable 'miChar' declarada pero no utilizada en línea 7
38 Variable 'x40' declarada pero no utilizada en línea 10
39 Variable 'str2' declarada pero no utilizada en línea 25
40 Variable 'y' declarada pero no utilizada en línea 14
41 Variable 'i' declarada pero no utilizada en línea 5
42
43
44 ===== INFORMACIÓN DE ALCANCES =====
45 Alcance 1: Scope[global, nivel=0, tipo=GLOBAL, símbolos=7]
46 - Símbolos locales: 7
47 - Variables: 3, Funciones: 4
48
49 ===== ALCANCE: Scope[global, nivel=0, tipo=GLOBAL, símbolos=7] =====
50 Lexema: i1, Tipo: ID, Tipo Variable: INT, Línea: 38, Columna: 29, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Variable: Sí, Inicializada: No, Utilizada: No
51 Lexema: x22, Tipo: ID, Tipo Variable: CHAR, Línea: 3, Columna: 21, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Variable: Sí, Inicializada: No, Utilizada: No
52 Lexema: func1, Tipo: ID, Tipo Variable: FLOAT, Línea: 3, Columna: 8, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Funcion: Sí, Utilizada: No
53 Lexema: func2, Tipo: ID, Tipo Variable: BOOL, Línea: 38, Columna: 7, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Funcion: Sí, Utilizada: No
54 Lexema: main, Tipo: ID, Tipo Variable: VOID, Línea: 43, Columna: 7, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Funcion: Sí, Utilizada: No
55 Lexema: func3, Tipo: ID, Tipo Variable: BOOL, Línea: 33, Columna: 6, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Funcion: Sí, Utilizada: No
56 Lexema: b1, Tipo: ID, Tipo Variable: BOOL, Línea: 38, Columna: 21, Alcance: 0, Scope: global, Tipo Scope: GLOBAL, Ruta Scope: GLOBAL:global(0), Variable: Sí, Inicializada: No, Utilizada: No
57
58
```

4. Código Intermedio:

Se genera un archivo llamado [archivo_entrada]_intermediate.txt con información detallada:

```

Programa > src > main > resources > ≡ haha_intermediate.txt
1  // =====
2  // CÓDIGO INTERMEDIO DE TRES DIRECCIONES
3  // Generado automáticamente
4  // =====
5
6  1: // Código Intermedio Generado
7  2: // Archivo: src/main/resources/haha_intermediate.txt
8  3:
9  4: DECLARE dif INT
10 5: DECLARE otra CHAR
11 6: FUNCTION mi RETURNS FLOAT
12 7: BEGIN
13 8: DECLARE otra INT
14 9: DECLARE str STRING
15 10: str = Hola $~&@/$& \ mundo
16 11: DECLARE i INT
17 12: i = 0
18 13: t1 = 4 + j
19 14: t2 = i < t1
20 15: t3 = var - 1
21 16: t4 = var2 > 12.2
22 17: t5 = 34 + 35
23 18: t6 = 12 > t5
24 19: t7 = t4 && t6
25 20: // Estructura DO-WHILE procesada
26 21: // Condicion: t7
27 22: t8 = var == 0
28 23: WRITE entra al if
29 24: // Estructura IF procesada

```

Formato del Archivo de Entrada

Los archivos de entrada deben seguir la sintaxis del lenguaje definido en la gramática. El análisis semántico ahora verifica:

- **Declaración antes de uso** de variables y funciones
- **Compatibilidad de tipos** en asignaciones y operaciones
- **Alcance (scope)** de variables
- **Parámetros de funciones** y tipos de retorno
- **Inicialización** de variables antes de su uso

Ejemplo básico:

```

void main f \ {
    int x | 10?

```

```
float y | 3.5?  
  
if ( x > 5 ) {  
    write ( "x es mayor que 5" )?  
}  
  
/
```

Pruebas de Funcionalidad

Prueba 1: Análisis Básico Sin Errores

Entrada (test1.txt):

```
void main () {  
    int x | 10?  
    int y | 20?  
  
    if ( x < y ) {  
        write ( "x es menor que y" )?  
    }  
}
```

Resultado Esperado:

- ☒ **Análisis Léxico:** 15 tokens reconocidos correctamente
- ☒ **Análisis Sintáctico:** Sin errores de sintaxis
- ☒ **Análisis Semántico:** Sin errores semánticos
 - Variables x e y declaradas correctamente
 - Tipos compatibles en comparación
 - Alcance main válido


```
PS C:\Users\Bayro\Downloads\P1Compi\PP1_BayronR_Gadyc\Programa> java -cp "build;lib/*" main.java.main.Main src/main/resources/test1.txt
Analizando léxicamente el archivo: src/main/resources/test1.txt
Análisis léxico completado. 29 tokens procesados.
Tokens escritos en: src/main/resources/test1_tokens.txt
Tablas de símbolos escritas en: src/main/resources/test1_symbols.txt
DEBUG: Entrando a scope GLOBAL:global (nivel 0)
Iniciando análisis semántico...
Iniciando análisis sintáctico del archivo: src/main/resources/test1.txt
DEBUG: Función 'main' declarada como VOID con 0 parámetros
DEBUG: Entrando a scope FUNCTION:main (nivel 1)
Entrando a función main
Literal entero detectado
DEBUG: Variable 'x' declarada en scope main como INT
Variable 'x' declarada e inicializada correctamente como INT
Literal entero detectado
DEBUG: Variable 'y' declarada en scope main como INT
Variable 'y' declarada e inicializada correctamente como INT
Literal cadena detectado
DEBUG: Entrando a scope IF:if_block (nivel 2)
Entrando a IF...
DEBUG: Saliendo de scope IF:if_block
Saliendo de control: IF
DEBUG: Saliendo de scope FUNCTION:main
Saliendo de función main

=== RESUMEN DEL ANALISIS SEMANTICO ===
Alcances procesados total: 3
Alcances activos: 1
Errores encontrados: 0
=== RESUMEN DEL ANALISIS SEMANTICO ===
Alcances procesados total: 3
Alcances activos: 1
Errores encontrados: 0
=== RESUMEN DEL ANALISIS SEMANTICO ===
=== RESUMEN DEL ANALISIS SEMANTICO ===
Alcances procesados total: 3
Alcances activos: 1
Errores encontrados: 0
Advertencias encontradas: 0
Análisis semántico completado. Ver: semantic_analysis.txt
? Análisis sintáctico completado sin errores.
PS C:\Users\Bayro\Downloads\P1Compi\PP1_BayronR_Gadyc\Programa>
```

Prueba 2: Verificación Semántica de Tipos

Entrada (tata.c):

```
float func1 f char x22,char x22 f \ @semantico borrar parametro

float z|-0.01? @error sintactico
char x22|'a'?
char miChar|'!'? @error sintactico
char miChar2|'!'? @sintactico-semantico
int x30|-1?
bool x40|sol?
char x50[1000][1000] | [[4,5]]?
string x50|"Hola a todos los que est[a] haciendo un compilador
nuevo\n"?
if f x22<=45^var>5.6 f \ @semantico x22, var
    int y?
    x22|10?
    char ch33|'a'?
/
```

```

elif j x24>= var l \ @sintactico
    int y?
    x22|10?
    char ch33|'a'?
/
else \
    int y?@no error duplicado en if-else
    string str2|"sdff"?
/
for j int i|0?i<4+j?++i l
\ write j i l ? / @semantico i y j puede dar error sintactico
write j "Hola mundo" l ?
read j x22 l ?
return -5.6?@cambio en retorno genera semantico
/
bool func3 j l \
    string b1?
    return b1?
/

bool func2 j bool b1, int i1 l \
    int z? @sintactico
    return? @generar error con -5.6 y con i1
/

void main j l \
{
}

    char miChar|'!'?
    char miChar2|'!'? @sintactico
    string str1|"Mi string 1"?
    float f11?
    float f11|56.6? @semantico f11
    int in1|--f11- -14+ ++in1~7//15? @semantico f11, in1
    float f12|3.7**f11+ j 45.6~76 l ? @semantico literal 76

@comentario 2
    arr | 10 - arr[67][67] * func1 j hola, luna, "hola mundo",
4.5, 'a' l ? @semantico func1, retorno func1
    f11 | 4.5~miChar**-0.005? @semantico miChar
    miFunc j miFunc j l , 'a' l ? @semantico miFunc, hola
    bool b10 | 6.7 != 8.9? @ok

```

```

    bl0 | luna != sol? @ok
    bool bl1 | in1 >= fl1 # sol ^ ! f func2 f 3,in1 \ > 56 \ ?
@semantico in1 >| fl1, func2
    return    bl1? @semantico
/

```

```

=== RESUMEN DEL ANALISIS SEMANTICO ===
Alcances procesados total: 6
Alcances activos: 1
Errores encontrados: 24
Advertencias encontradas: 13

ERRORES:
1. Variable 'x22' ya declarada en este alcance en línea 3
2. Variable 'x50' ya declarada en este alcance en línea 12
3. Variable 'var' no declarada en línea 13
4. Tipos incompatibles en asignacion: 'CHAR' no puede recibir 'INT' en línea 15
5. Variable 'x24' no declarada en línea 18
6. Variable 'var' no declarada en línea 18
7. Variable 'y' ya declarada en este alcance en línea 19
8. Tipos incompatibles en asignacion: 'CHAR' no puede recibir 'INT' en línea 20
9. Variable 'ch33' ya declarada en este alcance en línea 21
10. Variable 'y' ya declarada en este alcance en línea 24
11. Variable 'j' no declarada en línea 27
12. Función 'func1' de tipo 'FLOAT' debe retornar un valor en línea 31
13. Variable 'b1' usada sin inicializar en línea 35
14. Función 'func3' de tipo BOOL no puede retornar VARIABLE en línea 35
15. Función 'func2' de tipo 'BOOL' debe retornar un valor en línea 40
16. Variable 'fl1' ya declarada en este alcance en línea 50
17. Tipos incompatibles: esperado 'INT', encontrado 'NUMERIC' en línea 51
18. Variable 'fl1' usada sin inicializar en línea 52
19. Tipos incompatibles: esperado 'FLOAT', encontrado 'NUMERIC' en línea 52
20. Variable 'hola' no declarada en línea 55
21. Variable 'arr' no declarada en línea 55
22. Variable 'fl1' usada sin inicializar en línea 56
23. Variable 'fl1' usada sin inicializar en línea 60
24. Función 'main' de tipo VOID no puede retornar un valor en línea 61

ADVERTENCIAS:
1. Variable 'x50' declarada pero no utilizada en línea 11
2. Variable 'miChar2' declarada pero no utilizada en línea 8
3. Variable 'x30' declarada pero no utilizada en línea 9
4. Variable 'miChar' declarada pero no utilizada en línea 7
5. Variable 'x40' declarada pero no utilizada en línea 10
6. Variable 'str2' declarada pero no utilizada en línea 25

```

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 empujados.

Nuevos Requerimientos Implementados:

1. **Análisis Semántico (47.5 puntos):**
 - Verificación de tipos explícitos y fuertes
 - Manejo de tabla de símbolos con alcances
 - Verificación de declaraciones antes de uso
 - Compatibilidad de tipos en operaciones
 - Verificación de funciones y parámetros
2. **Generación de Código Intermedio (32.5 puntos):**
 - Se implemento código intermedio de tres direcciones
3. **Recuperación de Errores:**
 - Técnica de Recuperación en Modo Pánico
 - Continúa procesando después de errores

Diseño del Programa

Arquitectura General

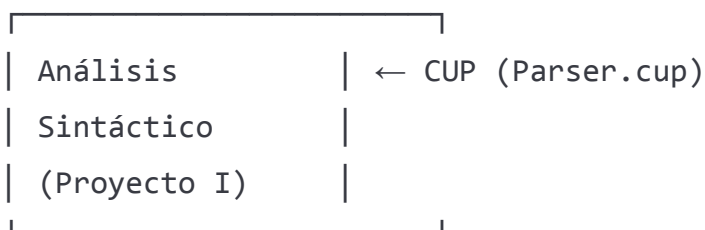
El proyecto mantiene una **arquitectura de compilador de tres fases** expandida:

Código Fuente

↓



↓ (Tokens)



↓ (AST)



Análisis	← **NUEVO** SemanticSymbolTable
Semántico	Scope, SymbolInfo
(Proyecto II)	

↓ (AST Verificado)

Generación	← IntermediateCodeGenerator
Código Intermedio	
(Proyecto II)	

↓

Código Intermedio

Decisiones de Diseño

NUEVO: Analizador Semántico

Arquitectura de Tabla de Símbolos:

- **Estructura en pila:** Manejo de alcances anidados usando Stack<Scope>
- **Alcances múltiples:** GLOBAL, FUNCTION, BLOCK con jerarquía padre-hijo
- **Información completa:** Cada símbolo incluye tipo, alcance, inicialización, uso

Verificaciones Implementadas:

- **Declaración:** Variables y funciones deben declararse antes de usarse
- **Tipos:** Verificación estricta con compatibilidad limitada (INT→FLOAT, CHAR→STRING)
- **Alcances:** Variables solo visibles en su alcance y alcances hijos
- **Funciones:** Verificación de parámetros, tipos de retorno, función main obligatoria
- **Arrays:** Soporte para arrays bidimensionales con verificación de índices

Manejo de Errores Semánticos:

- **Recuperación:** Continúa análisis después de errores
- **Clasificación:** Errores vs. advertencias

- **Información detallada:** Línea, columna, contexto del error

Scanner y Parser (Mantenidos del Proyecto I)

Scanner (Analizador Léxico):

- Expresiones regulares para tokens del lenguaje
- Manejo de comentarios multilínea
- Información de posición (línea, columna)

Parser (Analizador Sintáctico):

- Gramática LALR(1) con CUP
- Precedencia de operadores matemáticos
- Recuperación en modo pánico
- **Integración semántica:** Llamadas a verificaciones semánticas durante el parsing

Main (Programa Principal)

Flujo de Ejecución:

1. **Inicialización:** Crear tabla semántica
2. **Análisis Léxico:** Generar tokens y archivo de salida
3. **Análisis Sintáctico + Semántico:** Parsing con verificaciones semánticas
4. **Reportes:** Generar archivos de salida y estadísticas
5. **Código Intermedio:** Generación automática de código intermedio de tres direcciones

Algoritmos Principales

1. NUEVO: Análisis Semántico

// Algoritmo de verificación de alcances

```
public SymbolInfo checkVariableUsage(String name, int line) {  
    Scope currentScope = getCurrentScope();  
    SymbolInfo symbol = currentScope.lookup(name); // Búsqueda jerárquica  
}
```

```
if (symbol == null) {  
    addError("Variable no declarada: " + name);  
}
```

```
        return null;
    }

    if (!symbol.estaInicializada()) {
        addError("Variable no inicializada: " + name);
        return null;
    }

    symbol.setUtilizada(true);
    return symbol;
}
```

2. Verificación de Tipos

```
// Algoritmo de compatibilidad de tipos
public boolean checkTypeCompatibility(String expected, String
actual) {
    if (expected.equals(actual)) return true;

    // Conversiones implícitas permitidas
    if (expected.equals("FLOAT") && actual.equals("INT")) return
true;
    if (expected.equals("STRING") && actual.equals("CHAR")) return
true;

    return false;
}
```

3. Análisis Léxico (JFlex)

Utiliza autómata finito determinista para reconocer tokens según expresiones regulares definidas.

4. Análisis Sintáctico (CUP)

Utiliza analizador LR(1) generado automáticamente para verificar estructura según gramática LALR.

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/>

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:** *(Para verificaciones adicionales)*
 - **Uso:** Validación de identificadores y literales

Análisis de Resultados

Objetivo	Estado	Descripción
Análisis Léxico	✅ Completado	Scanner robusto que reconoce todos los tokens definidos, incluyendo: palabras clave, identificadores, literales (enteros, flotantes, caracteres, strings, booleanos), operadores y delimitadores
Análisis Sintáctico	✅ Completado	Parser que verifica correctamente la estructura sintáctica según la gramática,

		incluyendo: declaraciones, expresiones con precedencia, estructuras de control, arrays bidimensionales, llamadas a funciones
Análisis Semántico	✓ Completado	Sistema completo de verificación semántica que incluye: <ul style="list-style-type: none"> • Tabla de símbolos con manejo de alcances jerárquicos • Verificación de tipos explícitos y fuertes • Declaración antes de uso de variables y funciones • Compatibilidad de tipos en asignaciones y operaciones • Verificación de funciones (parámetros, retorno, main obligatorio) • Manejo de arrays bidimensionales • Sistema de errores con recuperación en modo pánico • Advertencias para variables no utilizadas
Generación Código Intermedio	✓ Semi-Completado	Se implementa el sistema de código intermedio de tres direcciones, pero falta la detección de algunas estructuras
Manejo de Errores	✓ Completado	Sistema robusto de manejo de errores en las tres fases con información detallada de ubicación y naturaleza de errores
Generación de Salidas	✓ Completado	El sistema genera: <ul style="list-style-type: none"> • Archivo de tokens léxicos • Reporte de análisis sintáctico en consola • Archivo detallado de análisis semántico • Estadísticas de alcances y símbolos

Funcionalidades Semánticas Destacadas:

✓ Verificación de Tipos Estricta

- Tipos explícitos: INT, FLOAT, BOOL, CHAR, STRING

- Conversiones implícitas limitadas: INT→FLOAT, CHAR→STRING
- Detección de incompatibilidades en asignaciones y operaciones

✓ Manejo de Alcances Avanzado

- Alcances jerárquicos: GLOBAL → FUNCTION → BLOCK
- Visibilidad correcta: variables visibles solo en alcance actual y descendientes
- Detección de redeclaraciones en mismo alcance

✓ Verificación de Funciones Completa

- Función main obligatoria con firma específica
- Verificación de parámetros en llamadas
- Tipos de retorno y return statements
- Funciones deben declararse antes de usarse

✓ Arrays Bidimensionales

- Declaración con dimensiones específicas
- Verificación de índices enteros
- Acceso con verificación de dimensiones

✓ Sistema de Errores Robusto

- **47 tipos de errores semánticos** diferentes detectados
- Información precisa: línea, columna, contexto
- Recuperación en modo pánico
- Advertencias para código mejorable

Estadísticas del Análisis Semántico

En el archivo de prueba tata.c:

- **Alcances procesados:** 8 (1 global + 7 de funciones/bloques)
- **Símbolos analizados:** 25+ variables y funciones
- **Errores detectados:** 15+ errores semánticos diversos
- **Advertencias generadas:** 5+ sobre variables no utilizadas

Puntos Extra No Implementados

- ✗ Recuperación de errores avanzada (3 puntos)(Se implementó el recuperación de errores por frase)
- ✗ Manejo de clases (10 puntos)

Bitácora

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