Instituto Tecnológico de Estudios Superiores de Monterrey, Campus Monterrey

Materia:

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)

Nombre de tarea: Entrega 2 (documentación) - módulo compiladores

Semestre: 8vo

Alumno:

Rafael Lorenzo Hernandez Zambrano A00836774

1) Diseño del Scanner y Parser

Proceso de elección para la herramienta automática del Parser y Scanner

A continuación, se realizará un análisis de las dos herramientas principales seleccionadas (JFlex + CUP y ANTLR) para el desarrollo en Java, con el fin de determinar su efectividad en la implementación del lenguaje 'Patito' (específicamente teniendo en cuenta las expresiones regulares y la gramática libre de contexto anteriormente realizada).

1) JFlex + CUP

Este enfoque, denominado "tradicional", es prácticamente análogo a Lex/Yacc en C. JFlex se encarga del análisis léxico, mientras que CUP gestiona el análisis sintáctico.
Las expresiones regulares previamente definidas poseen una sintaxis altamente compatible con JFlex, lo que facilita considerablemente su implementación.

Por otra parte, en lo que respecta a la gramática, la sintaxis también es compatible con CUP. Adicionalmente, la gramática ya se encuentra factorizada por precedencia, estableciendo una jerarquía. Este aspecto constituye un requisito fundamental para los parsers LALR(1) de tipo *bottom-up*, como CUP.

2) ANTLR

Al comparar las herramientas automáticas descritas previamente, se observa que esta herramienta ofrece un funcionamiento más simplificado al integrar el análisis léxico y sintáctico en un único archivo. Aunque ANTLR es una opción de análisis *top-down* en contraste con CUP, esta herramienta puede gestionar ambigüedades gramaticales, como la recursión izquierda, mediante la reescritura de la gramática. Su proceso de depuración es más accesible (humanamente legible) en comparación con los métodos *bottom-up*, como CUP. Consecuentemente, siendo esta una herramienta más moderna, esta logra compensar las desventajas de los métodos descendentes y capitalizar sus ventajas. Todo lo anterior, es gracias a su más sophisticado algoritmo LL(*).

Antes de determinar las herramientas automáticas a implementar, es crucial considerar la documentación que servirá de apoyo para el desarrollo del proyecto. Ambos conjuntos de herramientas disponen de una documentación exhaustiva; sin embargo, la de JFlex + CUP puede percibirse en ocasiones como desactualizada, mientras que la de ANTLR podría resultar excesivamente densa en relación con la magnitud y naturaleza del proyecto en cuestión.

Considerando las dos herramientas disponibles, se puede argumentar que, dada la compatibilidad de la gramática ‘Patito’ con parsers LALR(1), la naturaleza puramente académica y de tamaño reducido del proyecto (en comparación con iniciativas de ingeniería de software), la implementación de un método tradicional como **JFlex + CUP** en Java podría facilitar una aplicación más práctica de los conceptos teóricos abordados durante el curso. Además de una experimentación previa con su mismo funcionamiento en actividades pasadas por mi parte.

Desarrollo del Parser y Scanner

Teniendo en cuenta, la entrega pasada, mediante un procesador de texto se llegó a la siguiente definición del lenguaje ‘Patito’.

1) Expresiones regulares (RE)

- **Identificador (id):** `[a-zA-Z][a-zA-Z0-9_]*`
  - Un identificador debe comenzar con una letra, y puede ser seguido por letras o números.
- **Constante Entera (cte_int):** `[0-9]+`
  - Una constante entera consiste en uno o más dígitos consecutivos.
- **Constante Flotante (cte_float):** `[0-9]+\.[0-9]+`
  - Una constante flotante se compone de una parte entera inicial, seguida de un punto decimal y, posteriormente, la parte fraccional.
- **Constante de Cadena (cte_string):** `"[^"]*"`
  - Una constante de cadena consiste en cualquier secuencia de cero o más caracteres encerrados entre comillas dobles. Los caracteres internos no pueden ser comillas dobles.

2) Lista de Tokens

Palabras Reservadas:

- `program`
- `main`
- `end`
- `var`
- `int`
- `float`
- `void`
- `if`
- `else`
- `while`
- `do`
- `print`

Identificadores y Constantes:

- `ID` (Identificador)
- `cte_int` (Constante Entera)
- `cte_float` (Constante Flotante)
- `cte_string` (Constante de Cadena)

Operadores:

- = (Asignación)
- + (Suma)
- - (Resta)
- * (Multiplicación)
- / (División)
- > (Mayor que)
- < (Menor que)
- != (Diferente de)

Símbolos de Puntuación y Agrupación:

- ; (Punto y coma)
- , (Coma)
- : (Dos puntos)
- ((Paréntesis izquierdo)
-) (Paréntesis derecho)
- { (Llave izquierda)
- } (Llave derecha)

3) Reglas gramaticales (CFG)

Estructura General del Programa

- **PROGRAMA**: program ID ; **VARS FUNCS** main **BODY** end

Declaración de Variables

- **VARS**: var **LISTA_VARS** | ε
- **LISTA_VARS**: **DECLARACIÓN LISTA_VARS** | **DECLARACIÓN**
- **DECLARACIÓN**: **LISTA_IDS** : **TYPE** ;
- **LISTA_IDS**: ID , **LISTA_IDS** | ID
- **TYPE**: int | float

Definición de Funciones

- **FUNCS**: **DEF_FUNC FUNCS** | ε
- **DEF_FUNC**: **TIPO_RETORNO** ID (**PARAMS**) [**VARS**] [**BODY**] ;
- **TIPO_RETORNO**: void | **TYPE**
- **PARAMS**: **LISTA_PARAMS** | ε
- **LISTA_PARAMS**: ID : **TYPE** , **LISTA_PARAMS** | ID : **TYPE**

Cuerpo y Sentencias

- **BODY**: { **LISTA_SENTENCIAS** }
- **LISTA_SENTENCIAS**: **SENTENCIA LISTA_SENTENCIAS** | ε
- **SENTENCIA**: **ASSIGN** | **CONDITION** | **CYCLE** | **F_CALL** ; | **PRINT**

Definición de las Sentencias

- **ASSIGN**: ID = **EXPRESION ;**
- **PRINT**: print (**LISTA_PRINT**) ;
- **LISTA_PRINT**: **ELEMENTO_PRINT**, **LISTA_PRINT** | **ELEMENTO_PRINT**
- **ELEMENTO*PRINT: EXPRESION*** | ***CTE*_STRING**
- **CYCLE:** while (**EXPRESION**) do **BODY** ;
- **CONDITION: if (EXPRESION) BODY ELSE_PART** ;
- **ELSE_PART:** else **BODY** | ε

Jerarquía de expresiones

- **EXPRESION: EXP OP_RELACIONAL EXP** | **EXP**
- **OP_RELACIONAL:** > | < | !=
- **EXP: EXP** + **TÉRMINO** | **EXP** - **TÉRMINO** | **TÉRMINO**
- **TÉRMINO: TÉRMINO** * **FACTOR** | **TÉRMINO** / **FACTOR** | **FACTOR**
- **FACTOR:** (**EXPRESION**) | + **FACTOR** | - **FACTOR** | **CTE** | **F_CALL** | **ID**

Elementos atómicos

- **CTE:** CTE_INT **|** CTE_FLOAT
- **F_CALL:** ID (**ARGUMENTOS**)
- **ARGUMENTOS: LISTA_ARGUMENTOS** | ε
- **LISTA_ARGUMENTOS: EXPRESION** , **LISTA_ARGUMENTOS** | **EXPRESION**

La documentación anterior, será adaptada para su procesamiento con las herramientas automáticas seleccionadas. Recordando que el análisis léxico se desarrolló con la herramienta Jflex se puede observar en el archivo `Scanner.flex` y por el otro lado, análisis sintáctico se desarrolló con CUP dentro de `Parser.cup`.

Proceso de testing para el Scanner y el Parser

Para verificar la correcta implementación del lenguaje 'Patito', se llevó a cabo un riguroso proceso de *testing* tanto para el *Scanner* como para el *Parser*. Este proceso se realizó a través de la creación de archivos de prueba (`.txt`) que contienen código en 'Patito', abarcando diversas estructuras léxicas y sintácticas del lenguaje. El objetivo principal fue asegurar que ambas herramientas interpretan y procesan adecuadamente el código, identificando correctamente los tokens y validando la estructura gramatical según las reglas definidas.

A continuación a irá mostrando este proceso a través a distintos archivos de prueba (`.txt`):

Casos válidos (aquí se espera que el programa acepte los textos)

test_1.txt : Programa mínimo

```
program a;
main { }
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_1.txt
Iniciando análisis de: test_1.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

test_2.txt : Declaraciones de variables simples

```
program pvars;
var x : int;
main { }
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_2.txt
Iniciando análisis de: test_2.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

test_3.txt : Varias declaraciones y lista de ids

```
program TestExpresiones;
var
    a, b, c : int;
    x : float;
main
{}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_3.txt
Iniciando análisis de: test_3.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

test_4.txt : Declaración de funciones

```
program TestFuncs;
var g : int; // Variable global

// Función con parámetros, variables locales y cuerpo
int funccontodo(a:int, b:float)
var
    x : int;
{
    x = a + 1;
    print(x, b);
};

// Función sin parámetros y sin cuerpo
void funcsincuerpo();

// Función con cuerpo vacío
```

```
void funccuerpovacio()
{
};

main
{
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_4.txt
Iniciando análisis de: test_4.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

test_5.txt : Cobertura de sentencias

```
program TestStmts;
var
    x : int;
    y : float;

main
{
    // Asignación
    x = 10;
    y = 5.5;

    // Condición
    if (x > 5) {
        print("x es grande");
    };

    // Condición - con else
    if (y < 1.0) {
        print("y es chico");
    } else {
        print("y no es chico");
    };

    // Ciclo
    while (x > 0) do {
        x = x - 1;
    };

    // Impresión
    print("Valor final de x:", x, "Valor de y:", y + 1.0);

    // Llamada a función (f_call)
    // Asumiendo que 'miFunc' estuviera definida
    miFunc(x, y);
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_5.txt
Iniciando análisis de: test_5.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

test_6.txt : Manejo de expresiones

```
program TestExpr;
// comentario en una linea
var
```

```
    a, b, c : int;
    res : float;

/* Este es un comentario
   de bloque que abarca
   múltiples líneas.
*/

main
{
    a = 10;
    b = 5;
    c = 2;

    res = a + b * c; // res = 10 + (5 * 2) = 20.0

    res = (a + -b) * c; // res = (10 + -5) * 2 = 10.0

    if ( (a + b) != (c * 10) ) {
        print("Son diferentes");
    };
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_6.txt
Iniciando análisis de: test_6.txt
Análisis sintáctico completado con éxito.
Análisis finalizado.
```

Fin de la Sección de Casos Válidos

Con la ejecución de test_6.txt, se concluye la revisión de los casos de prueba válidos. Estos ejemplos han abarcado exitosamente todas las estructuras léxicas y sintácticas fundamentales que componen el lenguaje 'Patito', validando así la correcta interpretación del Scanner y Parser para construcciones bien formadas.

Casos inválidos (aquí se espera que el programa marque el error en las respectivas líneas)

test_1.txt : Error de estructura

```
program ErrorEstructura // <-- Falta ; aquí
var x : int;
main { }
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_1.txt
Iniciando análisis de: test_1.txt
[Error de Sintaxis] en línea 2, columna 1. Se encontró un token inesperado: (Tipo: VAR)
  > Syntax error
instead expected token classes are []
[Error de Sintaxis] en línea 2, columna 1. Se encontró un token inesperado: (Tipo: VAR)
  > Couldn't repair and continue parse
Error durante el análisis: Error fatal de sintaxis. No se puede continuar.
```

test_2.txt : Error en Declaración

```
program BadVar;
var
    a, b : int;
    x float; // <-- Falta :
    y : int; // La línea se procesa si la recuperación funciona
main { }
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_2.txt
Iniciando análisis de: test_2.txt
[Error de Sintaxis] en línea 4, columna 7. Se encontró un token inesperado: (Tipo: FLOAT)
  > Syntax error
instead expected token classes are []
[Recuperación] Error en declaración. Descartando hasta ';'.
Análisis completado con errores de sintaxis.
Análisis finalizado.
```

test_3.txt : Error en Sentencia

```
program BadStmt;
main
{
    x = 10 // <-- Falta ;
    y = 20;
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_3.txt
Iniciando análisis de: test_3.txt
[Error de Sintaxis] en línea 5, columna 5. Se encontró un token inesperado: 'y' (Tipo: ID)
  > Syntax error
instead expected token classes are [PLUS, MINUS, TIMES, DIV, GT, LT, NE, SEMI]
[Recuperación] Error en sentencia. Descartando hasta ';'.
Análisis completado con errores de sintaxis.
Análisis finalizado.
```

test_4.txt : Error en Expresión

```
program BadExpr;
main
{
    x = (10 + 5 * (2 + 3); // <-- Falta un ')'
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_4.txt
Iniciando análisis de: test_4.txt
[Error de Sintaxis] en línea 4, columna 26. Se encontró un token inesperado: (Tipo: SEMI)
  > Syntax error
instead expected token classes are []
[Recuperación] Error en sentencia. Descartando hasta ';'.
Análisis completado con errores de sintaxis.
Análisis finalizado.
```

test_5.txt : Carácter Inválido

```
program LexErrorChar;
main {
    x = 5 & 10; // <-- & no es un token válido
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_5.txt
Iniciando análisis de: test_5.txt
Error Léxico: Carácter no reconocido '&' en línea 3, columna 11
[Error de Sintaxis] en línea 3, columna 13. Se encontró un token inesperado: '10' (Tipo: CTE_INT)
  > Syntax error
instead expected token classes are [PLUS, MINUS, TIMES, DIV, GT, LT, NE, SEMI]
[Recuperación] Error en sentencia. Descartando hasta ';'.
Análisis completado con errores de sintaxis.
Análisis finalizado.
```

test_6.txt : String no Terminado

```
program Comment;
/* Este comentario
   nunca se cierra.
main { }
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_6.txt
Iniciando análisis de: test_6.txt
[Error de Sintaxis]. Se llegó al fin de archivo inesperadamente.
  > Syntax error
instead expected token classes are [INT, FLOAT, VOID]
[Error de Sintaxis]. Se llegó al fin de archivo inesperadamente.
  > Couldn't repair and continue parse
Error durante el análisis: Error fatal de sintaxis. No se puede continuar.
```

test_7.txt : Comentario de Bloque no Terminado

```
program UntermString;
main {
    print("Hola mundo); // <-- Falta " al final
}
end
```

Resultado obtenido:

```
(base) rafaellorenzohernandez@Mac Scanner and Parser % java -cp ".:java-cup-11b.jar" Main test_7.txt
Iniciando análisis de: test_7.txt
[Error de Sintaxis] en línea 3, columna 40. Se encontró un token inesperado: 'al' (Tipo: ID)
  > Syntax error
instead expected token classes are []
[Error de Sintaxis]. Se llegó al fin de archivo inesperadamente.
  > Couldn't repair and continue parse
Error durante el análisis: Error fatal de sintaxis. No se puede continuar.
```

Se debe recordar que todos los casos de prueba utilizados se encuentran de la carpeta `test_cases` dentro del código fuente, separados entre respuesta esperada: válida o inválida respectivamente. Además de las líneas necesarias para la compilación del programa se encuentran dentro de `README.md`.

Github con el código fuente:
https://github.com/rafhdz/proyecto-compiladores/tree/main/Scanner%20and%20Parser

2) Semántica de variables

Una vez validada la correcta estructura léxica y sintáctica del lenguaje "Patito" mediante las herramientas JFlex y CUP, el siguiente paso fundamental en el proceso de compilación es el análisis semántico.

Esta fase se encarga de verificar que el código, aunque sintácticamente correcto, tenga sentido lógico y cumpla con las reglas del lenguaje que no pueden ser expresadas únicamente con la gramática libre de contexto (CFG).
El objetivo principal es la gestión de tipos, la declaración y alcance (scope) de variables y funciones, y la coherencia entre tipos de datos y operaciones.

A continuación, se describen las estructuras de datos seleccionadas para esta tarea y las operaciones clave que se realizan sobre ellas.

Selección de Estructuras de Datos Semánticas

Para gestionar la información de estado del programa, se ha determinado el uso de tres estructuras de datos principales: el Directorio de Funciones, la Tabla de Variables y el Cubo Semántico respectivamente.

1. Directorio de Funciones (`FuncDir`)

Descripción:
Es la estructura de más alto nivel. Almacena todas las funciones declaradas en el programa, incluyendo dos ámbitos especiales:

- el scope global, tratado como una “función virtual” llamada "global", para almacenar variables globales.
- la función main, tratada como una función de tipo void con su propia tabla de variables locales.

Justificación (¿Por qué?):
La gramática permite múltiples funciones con tipo de retorno `(int, float o void)`, parámetros y variables locales.
Se requiere un directorio central para:

- Validar que no existan funciones duplicadas.
- Registrar la firma de cada función (tipo de retorno y lista ordenada de tipos de parámetros).
- Asociar cada función con su tabla de variables locales.

Implementación Seleccionada:
Una estructura `HashMap<String, FuncInfo>` donde la clave es el nombre de la función y el valor es un objeto `FuncInfo` que contiene:

- `tipoRetorno`: el tipo de retorno.
- `parametros`: una lista de parámetros.
- `VarTable`: la tabla de variables.

Esta implementación ofrece una complejidad promedio de $O(1)$ para inserciones y búsquedas.

2. Tabla de Variables (`VarTable`)

Descripción:
Cada función (incluyendo el `global y main`) posee su propia tabla de variables.
Esta tabla gestiona las variables declaradas dentro del ámbito actual.

Justificación (¿Por qué?):
La gramática define claramente la declaración de variables `(lista_ids : type;)`.
Se utiliza para:

- Validar duplicidad de nombres dentro de un mismo ámbito.
- Almacenar el tipo asociado a cada variable `(int o float)`.
- Consultar el tipo durante el uso de variables en expresiones y asignaciones.

Implementación Seleccionada:
Un `HashMap<String, VarInfo>` donde la llave es el nombre de la variable y el valor es un objeto `VarInfo` con:

- el tipo de dato (tipo)
- su dirección de memoria virtual (para fases posteriores)

Reglas de alcance (Scope):

- Si una variable no se encuentra en la tabla del ámbito actual, se busca en el ámbito "global".
- Si no existe en ninguno, se lanza el error: “Error: Variable no declarada.”

Al igual que el `FuncDir`, provee búsquedas en $O(1)$.

3. Cubo Semántico (`CuboSemantico`)

Descripción:
Estructura estática tridimensional que define las reglas de compatibilidad de tipos y operadores.
Cada celda del cubo representa el tipo de resultado (o error) de una operación entre dos operandos de ciertos tipos.

Justificación (¿Por qué?):
El lenguaje soporta operadores aritméticos (+, -, *, /), relacionales (>, <, !=) y de asignación (=).
El Cubo Semántico permite:

- Determinar el tipo resultante de una operación.
- Detectar incompatibilidades (ej. `int + string`).
- Centralizar y simplificar la lógica de tipos fuera del parser.

Reglas de compatibilidad implementadas:

- int y float son los únicos tipos numéricos.
- Las operaciones aritméticas devuelven float si alguno de los operadores es float.

- Las operaciones relacionales retorna un `int (1 o 0)`, ya que Patito no define un tipo boolean explícito.
- La asignación `float = int` es válida, pero `int = float` produce error (evita pérdida de precisión).

Implementación Seleccionada:
Una estructura `HashMap<String, HashMap<String, HashMap<String, String>>>`, consultada como:
`CuboSemantico[operador][tipoOperando1][tipoOperando2]`
Devuelve el tipo resultante o error si la combinación es inválida.
El acceso es en tiempo $O(1)$.

Operaciones Principales sobre las Estructuras

Las acciones semánticas (puntos neurálgicos) insertadas en el archivo Parser.cup ejecutarán las siguientes operaciones para construir y validar estas tablas:

Sobre el Directorio de Funciones (`FuncDir`):

Operación	Descripción
addFunction(nombre, tipoRetorno)	Se ejecuta al inicio del programa (global), al declarar funciones y al encontrar MAIN. Válida duplicados.
lookupFunction(nombre)	Se usa en llamadas a función. Si no existe, lanza: “Error: Función no declarada.”
addParam(funcNombre, tipoParam)	Registra cada parámetro formal en la lista de la función correspondiente y lo agrega a su VarTable.

A tener en cuenta: al finalizar el análisis de una función, el compilador regresa al ámbito global `(currentFunction = "global")`.

Sobre la Tabla de Variables (`VarTable`):

Operación	Descripción

addVariable(nombre, tipo)	Declara una nueva variable en el ámbito actual. Si ya existe, lanza “Error: Variable doblemente declarada.”
lookupVariable(nombre, ámbitoActual)	Busca una variable en el ámbito actual y, si no está, en el global. Si no existe, lanza “Error: Variable no declarada.”

Sobre el Cubo Semántico (`CuboSemantico`):

Operación	Descripción
checkOperation(operador, tipo1, tipo2)	Válida operaciones aritméticas o relacionales. Devuelve el tipo resultante o lanza “Error: Tipos incompatibles.”
checkAssignment(tipoTarget, tipoExpr)	Verifica compatibilidad de asignaciones. Permite float = int pero no int = float. Si es inválido, lanza “Error: Asignación incompatible.”

Integración por el código

Proceso de pruebas (comprar el funcionamiento del programa):

Prueba 1

```
program TestStmts;
var
    y : float;

main
{
    // Asignación
    x = 10;
    y = 5.5;

    // Condición
    if (x > 5) {
        print("x es grande");
    };
```

```
    // Condición - con else
    if (y < 1.0) {
        print("y es chico");
    } else {
        print("y no es chico");
    };

    // Ciclo
    while (x > 0) do {
        x = x - 1;
    };

    // Impresión
    print("Valor final de x:", x, "Valor de y:", y + 1.0);

    // Llamada a función (f_call)
    // Asumiendo que 'miFunc' estuviera definida
    miFunc(x, y);
}
end
```

Output:

```
[Error semántico] Variable 'x' no declarada.
Análisis sintáctico completado con éxito.

===== Estado Semántico =====
===== FuncDir =====
global -> { ret=void, params=[], vars={y={ tipo=float }} }
====================

Análisis completo sin errores.
```

Prueba 2

```
program TestExpr;
// comentario en una linea
var
    a, b, c : int;
    res : float;

/* Este es un comentario
   de bloque que abarca
   múltiples líneas.
*/

main
{
    a = 10;
    b = 5;
    c = 2;

    res = a + b * c; // res = 10 + (5 * 2) = 20.0

    res = (a + b) * c; // res = (10 + -5) * 2 = 10.0

    if ( (a + b) != (c * 10) ) {
        print("Son diferentes");
    };
}
end
```

Output:

```
Analizando archivo: programa.txt
Análisis sintáctico completado con éxito.

===== Estado Semántico =====
===== FuncDir =====
global -> { ret=void, params=[], vars={a={ tipo=int }, b={ tipo=int }, c={ tipo=int }, res={ tipo=float }} }
====================

Análisis completo sin errores.
```

Github respondiente:

https://github.com/rafhdz/proyecto-compiladores/tree/main/test