INSTITUTO TECNOLÓGICO DE COSTA RICA

ÁREA ACADÉMICA DE INGENIERÍA EN

COMPUTADORES

CE 1108 — Compiladores e Intérpretes

II Semestre

Investigación

Implementando un Intérprete

ESTUDIANTES:

Vindas Ortiz José María – 2022209471

Kun Kin Zheng Liang - 2022205015

PROFESOR:

Marco A. Hernandez Vasquez.

Septiembre, 2024

¿Cómo utilizar el análisis semántico y la tabla de símbolos dentro de un compilador?

El análisis semántico y la tabla de símbolos son fundamentales en la construcción de un compilador, ya que permiten verificar la correcta utilización de las estructuras y tipos de datos definidos en un programa fuente. A continuación, se explica cómo se utilizan estas herramientas para la comprobación de restricciones de tipo y otras limitaciones semánticas.

La tabla de símbolos es una estructura de datos que actúa como un registro de las declaraciones realizadas en el programa, como variables, funciones, y tipos. Cada entrada en la tabla de símbolos asocia un nombre (identificador) con información relevante, como su tipo y alcance. Esta información es utilizada durante el análisis semántico para verificar que el uso de cada identificador respete las reglas del lenguaje. Por ejemplo, si se declara una variable x como entera, la tabla de símbolos almacenará esta asociación y posteriormente, durante el análisis, cualquier operación realizada sobre x deberá ser compatible con el tipo entero (Ruslan's Blog, 2023).

El análisis semántico se encarga de verificar que las operaciones y el uso de las variables en el código sean coherentes con su definición. Para esto, se recorre el árbol de sintaxis abstracta (AST) y se realiza una verificación de tipo en cada nodo. Por ejemplo, en una expresión de adición, se debe comprobar que ambos operandos son de tipos compatibles (por ejemplo, ambos enteros o ambos flotantes). Esto se logra mediante métodos recursivos que verifican los subárboles del AST y acumulan información de tipo en la tabla de símbolos (Cornell University, 2023).

Para gestionar el alcance de las variables, se utiliza un contexto de tipos, que puede representarse como una pila de tablas de símbolos. Cada vez que se entra en un nuevo ámbito (por ejemplo, al iniciar una función), se crea un nuevo contexto que hereda el contexto anterior. Al salir del ámbito, el contexto se elimina. Esta estructura permite rastrear los tipos de las variables incluso cuando se utilizan en diferentes ámbitos anidados (Cornell University, 2023).

La comprobación de restricciones semánticas incluye la verificación de que no se realicen operaciones inválidas, como sumar un número y un booleano. Durante el

análisis, cada operación se evalúa según las reglas del lenguaje y cualquier infracción genera un error semántico. Por ejemplo, una expresión que intente asignar un valor flotante a una variable entera debe ser señalada como error (Ruslan's Blog, 2023).

Por lo tanto, la tabla de símbolos y el análisis semántico permiten asegurar que un programa fuente sea semánticamente válido antes de proceder a su ejecución o traducción a código máquina. Estas herramientas son críticas para garantizar que el programa se comporte como se espera y para evitar errores de ejecución que puedan surgir de inconsistencias en el uso de tipos y variables.

Principales Hallazgos

Durante el desarrollo de un compilador o intérprete, una de las etapas cruciales es el análisis semántico. Esta etapa sigue al análisis sintáctico, donde el código fuente se descompone en un árbol sintáctico que refleja la estructura del programa. A partir de este árbol, el análisis semántico se encarga de verificar que las construcciones del lenguaje respeten las reglas del lenguaje, como la coherencia de tipos de datos, el alcance de las variables y las operaciones permitidas para cada tipo.

Para realizar el análisis semántico de manera eficiente, es fundamental utilizar una tabla de símbolos. Esta tabla actúa como un diccionario que asocia nombres de variables, funciones y otros identificadores con su información relevante, como tipo de dato, valor actual, y ámbito de visibilidad. La tabla de símbolos es esencial para detectar errores como el uso de variables no declaradas, conflictos de tipos en operaciones o el intento de usar funciones con argumentos incorrectos.

En la implementación de nuestro intérprete, se utilizó una gramática de atributos. Esta técnica extiende las reglas sintácticas con símbolos adicionales que permiten la ejecución de rutinas semánticas durante el análisis. Cada regla de la gramática se complementa con acciones específicas que se ejecutan cuando la regla se cumple. Por ejemplo, cuando se detecta una operación aritmética, se verifica que los operandos sean del tipo adecuado y que la operación sea válida para esos tipos.

Además, se implementó el patrón Interpreter para la evaluación de expresiones. Este patrón permite definir una clase para cada tipo de operación o expresión en el lenguaje, encapsulando la lógica de evaluación en cada clase. Esto facilita la extensión del lenguaje, permitiendo agregar nuevas operaciones simplemente creando nuevas clases.

Un hallazgo importante durante este proceso fue la necesidad de manejar adecuadamente las operaciones de comparación y lógica. Para estas operaciones, se debe asegurar que ambos operandos sean del mismo tipo y que el tipo resultante sea booleano. En caso de un error, se utiliza una propagación de errores mediante una excepción, lo que permite al intérprete detectar y reportar problemas sin detener su ejecución abruptamente.

Otro aspecto relevante fue la generación de representaciones intermedias del código. Estas representaciones facilitan la optimización y la posterior generación de código objeto o interpretación directa. En nuestro caso, se utilizaron cuádruplas, que descomponen cada operación en una estructura simple (operador, operando1, operando2, resultado) que puede ser evaluada o transformada fácilmente.

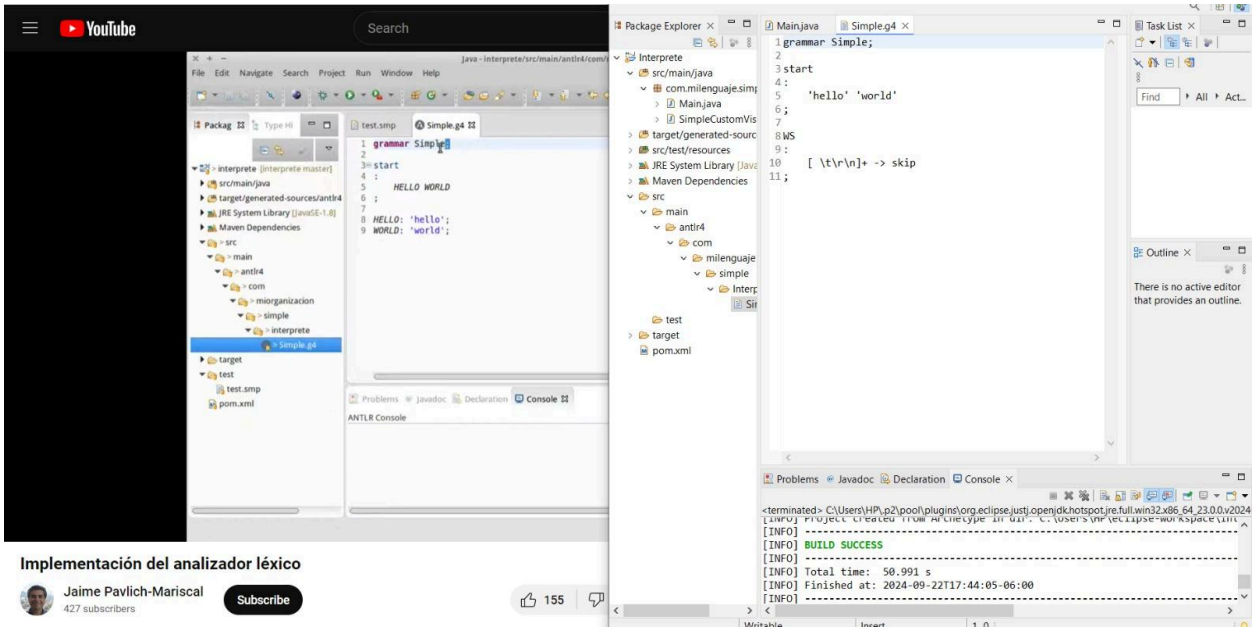
Finalmente se tiene que tener en cuenta que al momento de crear una gramática regular, es importante el orden en que uno coloque los tokens porque, en el caso de ANTLR, revisa de forma secuencial, se encontró que las funciones más usadas se deben implementar lo más arriba posible y las menos usadas se deben implementar lo mas abajo, asi ahorrando tiempo en ejecuciones.

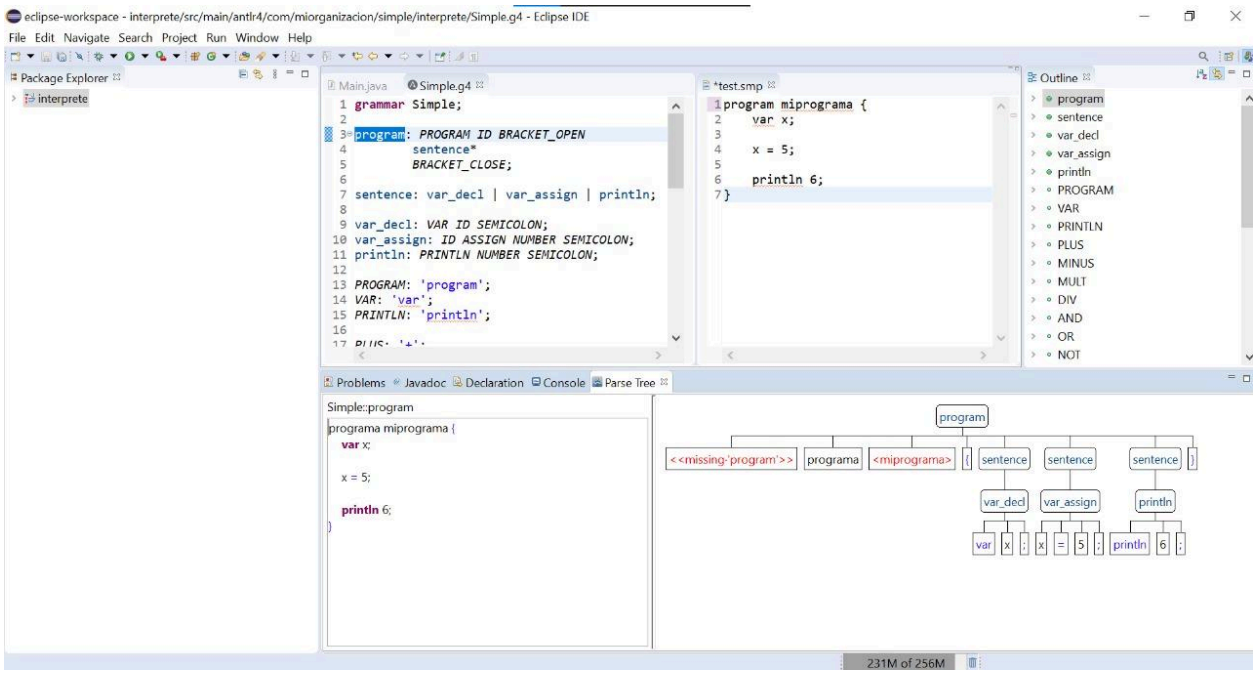
En resumen, el uso del análisis semántico y la tabla de símbolos dentro del intérprete permitió no solo validar el código fuente de manera rigurosa, sino también extender el lenguaje de manera modular y eficiente. La implementación del patrón Interpreter simplificó la estructura del intérprete, haciendo el código más mantenible y permitiendo pruebas exhaustivas de cada componente del lenguaje.

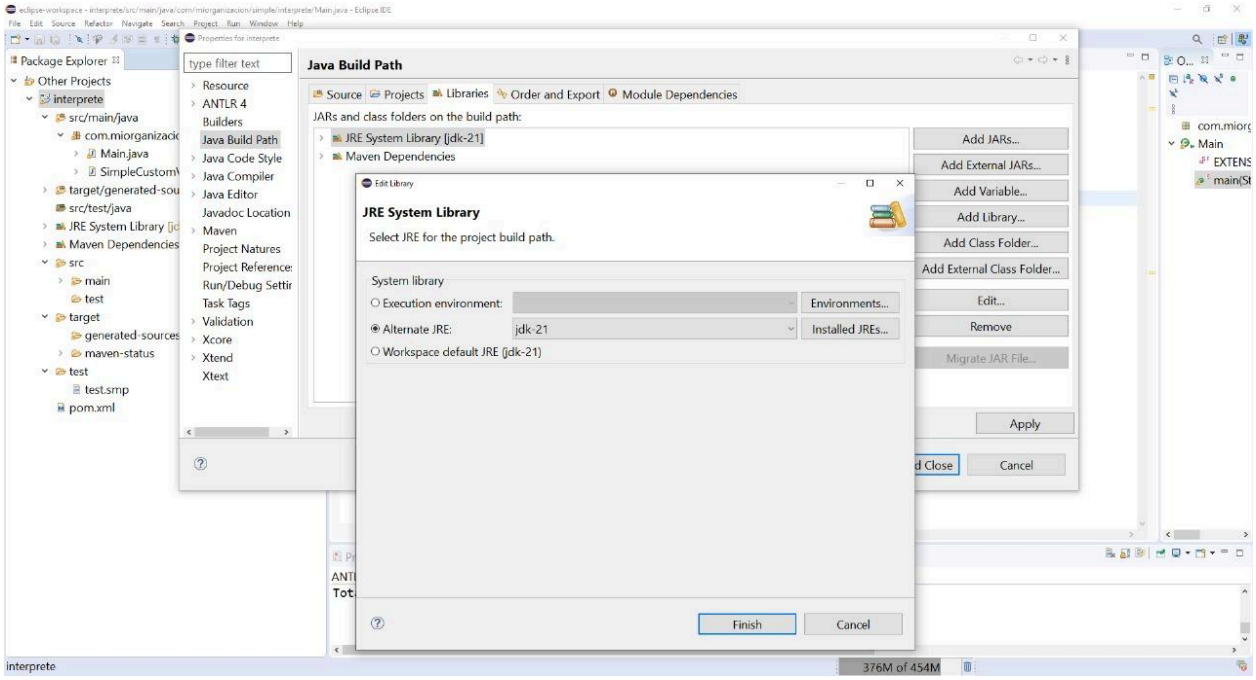
Estos hallazgos y técnicas fueron implementados con éxito, logrando un intérprete capaz de procesar correctamente expresiones aritméticas, condicionales y sentencias de impresión, respetando las reglas semánticas del lenguaje definido.

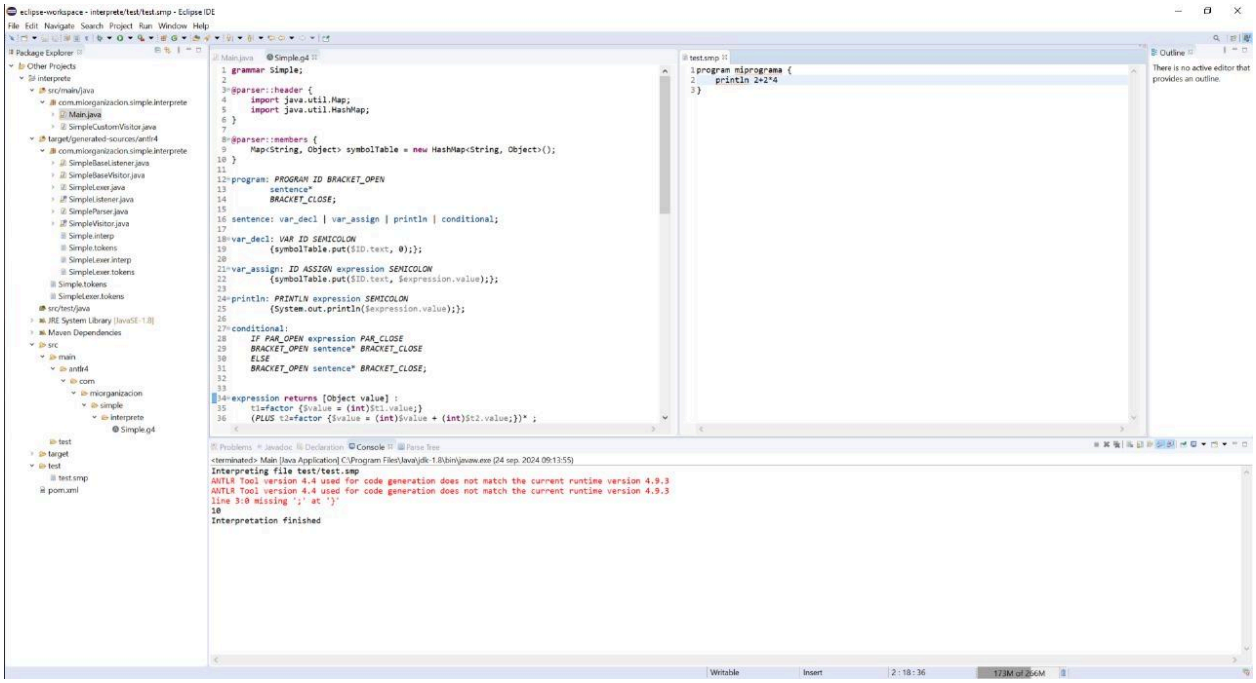
Evidencia de la implementación del intérprete

Pavlich-Mariscal (2016) implementa una guía para el desarrollo de un intérprete empleando ANTLR4, el cual se toma como base y las evidencias de su ejecución de muestra a continuación:






Java Build Path
Source
Projects
Libraries
Order and Export
Module Dependencies
JARs and class folders on the build path:
JRE System Library [jdk-21]
Maven Dependencies
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
Apply
Cancel
JRE System Library
Select JRE for the project build path.
System library
Execution environment:
Environments...
Alternate JRE:
jdk-21
Installed JREs...
Workspace default JRE (jdk-21)
Finish
Cancel
376M of 454M


Interpreting file test/test.smp
ANTLR Tool version 4.4 used for code generation does not match the current runtime version 4.9.3
ANTLR Tool version 4.4 used for code generation does not match the current runtime version 4.9.3
line 3:0 missing ';' at '}'
10
Interpretation finished

Modificación implementada

Se decidió implementar la operación pendiente en el programa siendo este la de la división, primeramente implementamos que pueda leer el carácter correspondiente, en este caso "/".

```
80 PLUS: '+';
81 MINUS: '-';
82 MULT: '*';
83 DIV: '/';
```

Seguidamente, se implementó su respectivo proceso de derivación del programa a través del "divb", debido a que la lógica del división es similar a la de la multiplicación, se implemento de similar manera.

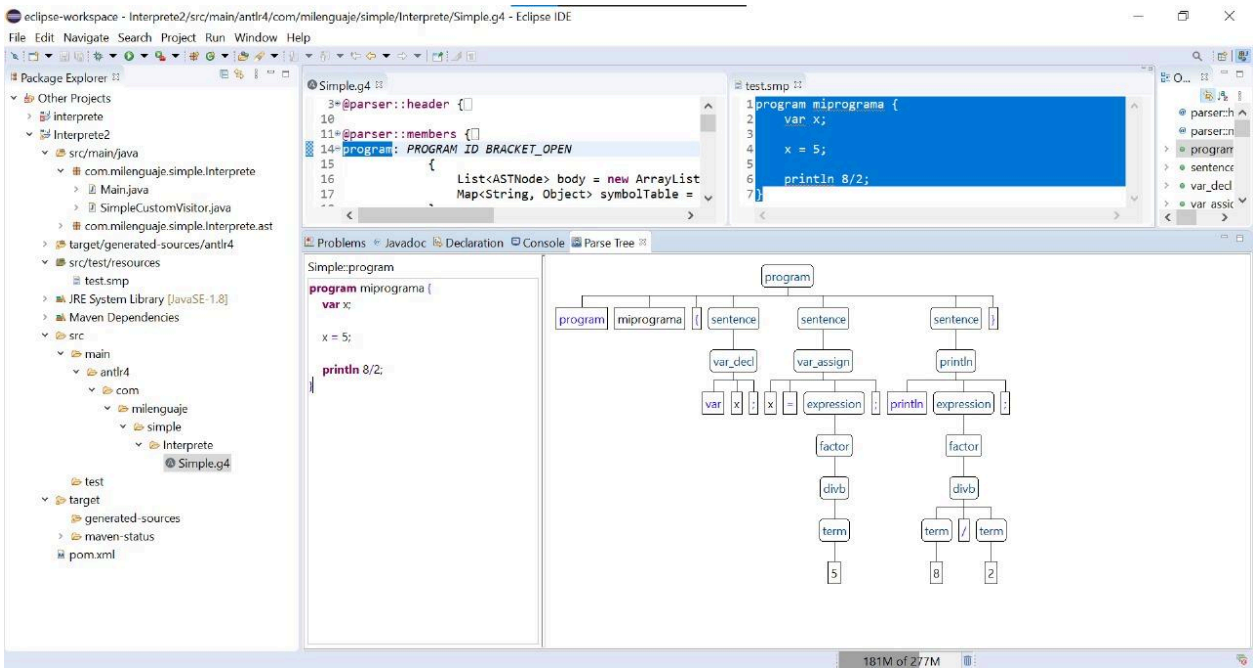
```
60 factor returns [ASTNode node]:
61     t1=divb {$node = $t1.node;}
62         (MULT t2=divb {$node = new Multiplication($node, $t2.node);})*;
63
64 divb returns [ASTNode node]:
65     t1=term {$node = $t1.node;}
66         (DIV t2=term {$node = new Division($node, $t2.node);})*;
67
68 term returns [ASTNode node]:
69     NUMBER {$node = new Constant(Integer.parseInt($NUMBER.text));}
70     | ID {$node = new VarRef($ID.text);}
71     | BOOLEAN {$node = new Constant(Boolean.parseBoolean($BOOLEAN.text));}
72     | PAR_OPEN expression {$node = $expression.node;} PAR_CLOSE;
73
```

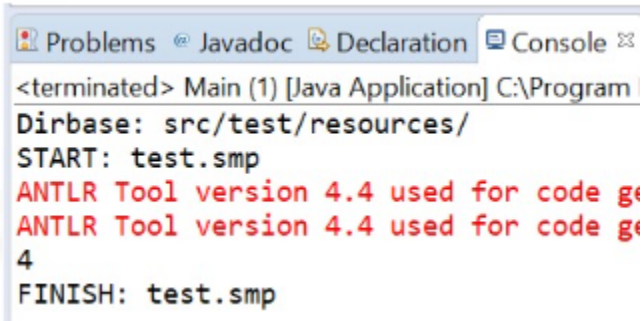
Finalmente, se desarrolló una clase que heredó de ASTNode, la cual se muestra a continuación:

Simple.g4 test.smp Main.java Division.java

```java
package com.milenguaje.simple.Interprete.ast;

import java.util.Map;

public class Division implements ASTNode {
    private ASTNode operand1;
    private ASTNode operand2;

    public Division(ASTNode operand1, ASTNode operand2) {
        super();
        this.operand1 = operand1;
        this.operand2 = operand2;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        return (int)operand1.execute(symbolTable) / (int)operand2.execute(symbolTable
    }

}
```

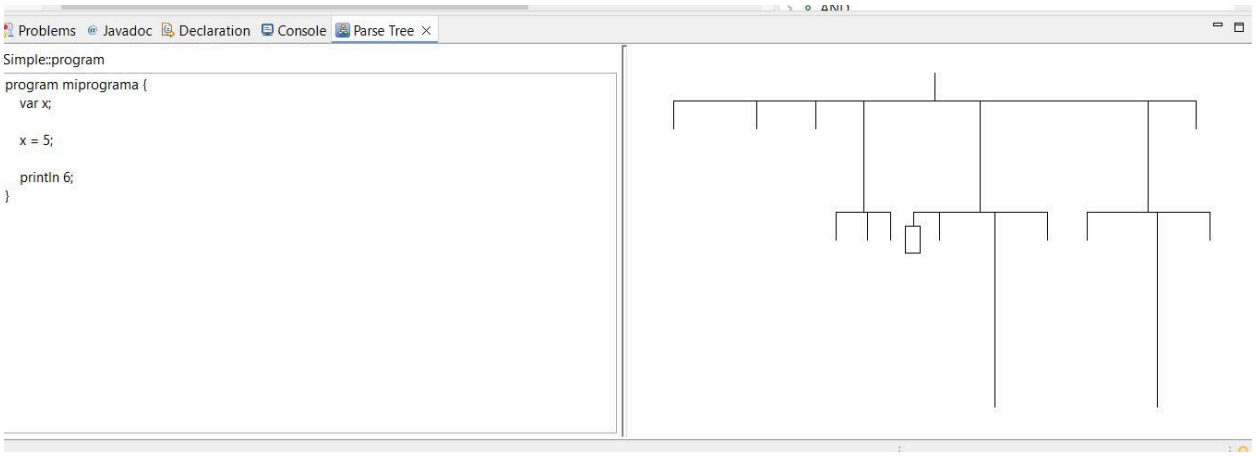
Como resultado, se obtuvo la implementación adecuada de la división en el programa al obtener el resultado deseado en consola, y un Árbol de Parseo concordante con lo esperado.





Errores encontrados

Un error bastante frecuentado a lo largo de la implementación del intérprete consiste en que a la hora de la generación del árbol este no llegaba a mostrar valores, ninguno, solamente ramificaciones, tal como se muestra continuación:



No se encontró alguna razón en específico para la generación del problema; se concluyó que el error podría consistir en las diferentes versiones de herramientas y programas, mostrando en la terminal la salida esperada, es decir, hacía la impresión del valor normal, sin embargo, no llegaba a mostrar el árbol. Cabe mencionar que no se cuenta con total seguridad para afirmar esto como la razón del problema.

```
Dirbase: src/test/resources/
START: test.smp
ANTLR Tool version 4.4 used for code generation does not match the current runtime version 4.9.2
ANTLR Tool version 4.4 used for code generation does not match the current runtime version 4.9.2
6
FINISH: test.smp
```

Por otro lado, en compilación se presenta un error en la aplicación de ANTLR4 cuando se cambia seguidamente la sección semántica o léxica y genera errores, este puede presentar los errores incluso después de corregirlos, provocando incluso problemas de compilado, aun si ya no existen los errores.

Description	Resource	Path	Location	Type
Errors (8 items)				
attribute node isn't a valid property in $ID.node (org.antlr:antlr4-mav	pom.xml	/Interprete	line 38	Maven Build Pa...
rule LEFT redefinition; previous at line 105 (org.antlr:antlr4-maven-plu	pom.xml	/Interprete	line 107	Maven Build Pa...
rule program contains a closure with at least one alternative that can	pom.xml	/Interprete	line 14	Maven Build Pa...
rule program contains a closure with at least one alternative that can	pom.xml	/Interprete	line 15	Maven Build Pa...
unknown attribute node for rule add in $add.node (org.antlr:antlr4-m	pom.xml	/Interprete	line 32	Maven Build Pa...
unknown attribute reference NUMBER in $NUMBER.text (org.antlr:ant	pom.xml	/Interprete	line 106	Maven Build Pa...
unknown attribute reference NUMBER in $NUMBER.text (org.antlr:ant	pom.xml	/Interprete	line 107	Maven Build Pa...

Una de las soluciones posibles es volver a compilar desde el principio o crear un nuevo programa ANTLR4 y copiar el código ya corregido en el nuevo programa, para cargar correctamente el programa.

Otro desafío importante fue la incompatibilidad con el JDK. Durante el desarrollo, surgieron problemas debido a que el proyecto requería una versión específica de JDK que no estaba instalada en el entorno de trabajo. Esto llevó a errores en la compilación y ejecución, lo que exigió configurar correctamente el entorno de desarrollo para que utilizara la versión adecuada del JDK.

Finalmente, las excepciones NullPointerException representaron un obstáculo frecuente. Estas excepciones ocurren cuando se intenta utilizar un objeto que no ha sido inicializado, lo cual es un error común en la programación. Identificar y resolver estas excepciones implicó revisar detalladamente el flujo del programa y asegurar que todas las referencias a objetos se inicializaran correctamente antes de su uso.

Referencias Bibliográficas

Cornell University. (2023). *Type Checking and Symbol Tables in Compilers*. Recuperado de Cornell University

Pavlich-Mariscal, J. (2016, 8 de abril). Proceso de interpretación de un lenguaje de programación [Vídeo]. YouTube. https://www.youtube.com/watch?v=WrlgULIJqEw

Ruslan's Blog. (2023). *Let's Build A Simple Interpreter. Part 13: Semantic Analysis*. Recuperado de Ruslan's Blog