Ingeniería en Computadores

Compiladores e Interpretes

Implementando un Interprete

Investigación y trabajo en campo

Profesor:

Marco Hernández Vásquez

Integrantes:

Giancarlo Vega Marín

Sebastián Chaves Ruiz

Implementación del intérprete

El intérprete fue implementado utilizando ANTLR4.
La gramática desarrollada define tanto las reglas sintácticas del lenguaje como la lógica para construir el árbol de sintaxis abstracta (AST) y posteriormente ejecutar cada nodo.

Se desarrolló un parser denominado ***Simple***, el cual amplía las funcionalidades de ANTLR. En este se incluyó un bloque ***@parser::header*** para importar las librerías necesarias de Java, así como las clases que representan los nodos del árbol de sintaxis abstracta (AST). Además, en el bloque ***@parser::members*** se estableció una tabla de símbolos global, implementada como un ***HashMap<String, Object>***, que almacena los identificadores y sus valores a lo largo de la ejecución.

En cuanto a las reglas, la principal es la correspondiente a ***program***, que reconoce la estructura básica de un programa.

Se creó la regla **sentence**, que contempla distintos tipos de instrucciones, como **println**, **conditional**, **var_decl** y **var_assign**. Cada una de ellas retorna un nodo del AST, el cual conoce su propio modo de ejecución.

El lenguaje también contempla reglas para las **expresiones**, que abarcan tanto operaciones aritméticas como lógicas, estas se modelan mediante nodos específicos que implementan el método **execute(symbolTable)**.

En el nivel más básico, los **factores y términos** pueden ser números, booleanos, identificadores o expresiones agrupadas entre paréntesis.

Ademas se definieron los **tokens**, que comprenden las palabras reservadas como program, if, else, los operadores como +, *, &&, fibsum, los identificadores, los valores numéricos y los booleanos.

Reglas principales

program

```
program
    : PROGRAM ID BRACKET_OPEN
        {
            List<ASTNode> body = new ArrayList<ASTNode>();
        }
        (s=sentence { body.add($s.node); })*
      BRACKET_CLOSE
        {
            for (ASTNode n : body) {
                n.execute(symbolTable);
            }
        }
    ;
```

Reconoce la estructura base de un programa, dentro de las llaves se almacenan las sentencias en una lista (body) y posteriormente se ejecutan una a una.

sentence

```
sentence returns [ASTNode node]
    : println     { $node = $println.node; }
    | conditional { $node = $conditional.node; }
    | var_decl    { $node = $var_decl.node; }
    | var_assign  { $node = $var_assign.node; }
    ;
```

Define los posibles tipos de sentencias: println, conditional, var_decl y var_assign, además cada sentencia devuelve un nodo del AST que sabe ejecutarse por sí mismo.

println

```
println returns [ASTNode node]
    : PRINTLN expression SEMICOLON
        { $node = new Println($expression.node); }
    ;
```

Implementa la instrucción de salida esta recibe una expresión y la imprime.

conditional

```
conditional returns [ASTNode node]
    : IF PAR_OPEN expression PAR_CLOSE
        {
            List<ASTNode> body = new ArrayList<ASTNode>();
        }
        BRACKET_OPEN (s1=sentence { body.add($s1.node); })* BRACKET_CLOSE
        ELSE
        {
            List<ASTNode> elseBody = new ArrayList<ASTNode>();
        }
        BRACKET_OPEN (s2=sentence { elseBody.add($s2.node); })* BRACKET_CLOSE
        {
            $node = new If($expression.node, body, elseBody);
        }
    ;
```

Implementa estructuras if – else, construyendo dos listas de sentencias una para el bloque if y otra para el bloque else.

var_decl y var_assign

```
var_decl returns [ASTNode node]
    : VAR ID SEMICOLON
        { $node = new VarDecl($ID.text); }
    ;

var_assign returns [ASTNode node]
    : ID ASSIGN expression SEMICOLON
        { $node = new VarAssign($ID.text, $expression.node); }
    ;
```

Permiten declarar variables y asignarles valores, encargados de modificar la tabla de símbolos.

expression

```
expression returns [ASTNode node]
    : t1=factor { $node = $t1.node; }
      (PLUS t2=factor { $node = new Addition($node, $t2.node); })*
      (AND t3=factor { $node = new And($node, $t3.node); })*
      (POW2ADD t2=factor { $node = new Pow2Add($node, $t2.node); })*
      (FIBSUM t2=factor { $node = new FibSum($node, $t2.node); })*
    ;
```

Permite manejar tanto operaciones aritméticas como lógicas, tal como Addition, Multiplication, And, Pow2Add, FibSum.

logic_expr

```
logic_expr returns [ASTNode node]
    : t1=factor { $node = $t1.node; }
      (AND t2=factor { $node = new And($node, $t2.node); })*
    ;
```

Esta regla permite construir expresiones lógicas utilizando el operador AND &&

arithExpr

```
arithExpr returns [ASTNode node]
    : t1=factor { $node = $t1.node; }
      (PLUS t2=factor { $node = new Addition($node, $t2.node); })*
    ;
```

Esta regla define expresiones aritméticas simples basadas en la suma.

factor

```
factor returns [ASTNode node]
    : t1=term { $node = $t1.node; }
      (MULT t2=term { $node = new Multiplication($node, $t2.node); })*
    ;
```

Permite la multiplicación de un solo término o de varios términos. Cada vez que se encuentra un operador de multiplicación, se genera un nodo Multiplication en el árbol de sintaxis abstracta, que combina el factor anterior con el nuevo.

Este permite manejar correctamente las expresiones con multiplicaciones encadenadas, respetando la precedencia de los operadores.

term

```
term returns [ASTNode node]
    : NUMBER  { $node = new Constant(Integer.parseInt($NUMBER.text)); }
    | BOOLEAN { $node = new Constant(Boolean.parseBoolean($BOOLEAN.text)); }
    | ID      { $node = new VarRef($ID.text); }
    | PAR_OPEN e=expression PAR_CLOSE { $node = $e.node; }
    ;
```

Esta regla representa la unidad más básica dentro de un factor “termino” .

Puede ser:

- Un número, que se transforma en un nodo Constant de tipo entero.

- Un valor booleano, como true o false, que se convierte en un nodo Constant de tipo booleano.
- Un identificador, es decir, el nombre de una variable, que se modela mediante un nodo VarRef encargado de consultar su valor en la tabla de símbolos.
- Una expresión encerrada entre paréntesis, que permite agrupar operaciones y asegurar que se respeten las prioridades de los operadores.

Tokens léxicos del lenguaje

Los tokens definen los elementos básicos que el analizador léxico (lexer) reconoce en el código fuente.

El lenguaje define palabras reservadas como program, var, println, if y else para estructurar programas y controlar el flujo.

```
PROGRAM: 'program';
VAR: 'var';
PRINTLN: 'println';
IF: 'if';
ELSE: 'else';
```

Incluye operadores aritméticos (+, -, *, /), lógicos (&&, ||, !) y de comparación (>, <, >=, <=, ==, !=), además de operaciones especiales como Pow2Add y fibsum.

```
PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';
POW2ADD: '**+';
FIBSUM: 'fibsum';
```

El operador = se usa para asignación.

```
ASSIGN: '=';
```

Los delimitadores { }, ( ) y ; organizan bloques y expresiones. Los tipos de datos son BOOLEAN (true/false), NUMBER (enteros) e ID (identificadores).

```
BRACKET_OPEN: '{';
BRACKET_CLOSE: '}';

PAR_OPEN: '(';
PAR_CLOSE: ')';

SEMICOLON: ';';

BOOLEAN: 'true' | 'false';

ID: [a-zA-Z_][a-zA-Z0-9_]*;

NUMBER: [0-9]+;
```

Los espacios, tabulaciones y saltos de línea (WS) se ignoran en la ejecución.

```
WS: [ \t\n\r]+ -> skip;
```

Clases AST
Addition

La clase Addition representa la operación de suma dentro del intérprete. Cada vez que se encuentra una expresión con el operador +, se crea un nodo Addition en el AST que guarda los dos operandos de la suma.

```java
package ast;

import java.util.Map;

public class Addition implements ASTNode {  no usages  Sebas36762
    private ASTNode operand1;  2 usages
    private ASTNode operand2;  2 usages
    public Addition(ASTNode operand1, ASTNode operand2) {  2 usages  Sebas36762
        super();
        this.operand1 = operand1;
        this.operand2 = operand2;
    }


    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        return (int)operand1.execute(symbolTable) + (int)operand2.execute(symbolTable);
    }


}
```

ASTNode

La interfaz ASTNode define el contrato básico que deben cumplir todos los nodos del árbol de sintaxis abstracta (AST) del intérprete. Cada clase que implementa ASTNode debe proveer el método execute, que recibe la tabla de símbolos y devuelve el resultado de evaluar ese nodo.

```java
package ast;

import java.util.Map;

public interface ASTNode {  Sebas36762
    public Object execute(Map<String, Object> symbolTable);
}
```

Constant

La clase Constant representa un **valor literal** en el intérprete, como un número o un booleano. Cada vez que se encuentra un valor fijo en una expresión, se crea un nodo Constant que guarda ese valor.

```java
package ast;

import java.util.Map;

public class Constant implements ASTNode {  no usages  Sebas36762
    private Object value;  2 usages
    public Constant(Object value) {  2 usages  Sebas36762
        super();
        this.value = value;
    }
    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) { return value; }
}
```

If

La clase If representa una **estructura condicional** en el intérprete. Cada nodo If guarda una **condición** y dos listas de sentencias: una para el bloque if y otra para el bloque else.

```java
public class If implements ASTNode {  no usages  Sebas36762
    private ASTNode condition;  2 usages
    private List<ASTNode> body;  2 usages
    private List<ASTNode> elseBody;  2 usages

    public If(ASTNode condition, List<ASTNode> body, List<ASTNode> elseBody) {
        super();
        this.condition = condition;
        this.body = body;
        this.elseBody = elseBody;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        if ((boolean)condition.execute(symbolTable)) {
            for (ASTNode n : body) {
                n.execute(symbolTable);
            }
        } else {
            for (ASTNode n : elseBody) {
                n.execute(symbolTable);
            }
        }
        return null;
    }
```

Multiplication

La clase Multiplication representa la operación de **multiplicación** dentro del intérprete. Cada vez que se encuentra un operador *, se crea un nodo Multiplication en el AST que guarda los dos operandos de la multiplicación.

```java
package ast;

import java.util.Map;

public class Multiplication implements ASTNode {  no usages  Sebas36762
    private ASTNode operand1;  2 usages
    private ASTNode operand2;  2 usages
    public Multiplication(ASTNode operand1, ASTNode operand2) {  1 usage  Sebas36762
        super();
        this.operand1 = operand1;
        this.operand2 = operand2;
    }


    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        return (int)operand1.execute(symbolTable) * (int)operand2.execute(symbolTable);
    }
}
```

VarAssign

La clase VarAssign representa la **asignación de valores a variables** en el intérprete. Cada nodo VarAssign guarda el nombre de la variable y la expresión cuyo valor se quiere asignar.

```java
package ast;

import java.util.Map;

public class VarAssign implements ASTNode {  no usages  Sebas36762
    private String name;  2 usages
    private ASTNode expression;  2 usages

    public VarAssign(String name, ASTNode expression) {  1 usage  Sebas36762
        super();
        this.name = name;
        this.expression = expression;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        symbolTable.put(name, expression.execute(symbolTable));
        return null;
    }
}
```

VarDecl

La clase VarDecl representa la **declaración de una variable** en el intérprete. Cada nodo VarDecl guarda el nombre de la variable que se quiere declarar.

```java
package ast;

import java.util.Map;

public class VarDecl implements ASTNode {  no usages  Sebas36762
    private String name;  2 usages

    public VarDecl(String name) {  1 usage  Sebas36762
        super();
        this.name = name;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        symbolTable.put(name, new Object());
        return null;
    }
}
```

VarRef

La clase VarRef representa una **referencia a una variable** dentro del intérprete. Cada nodo VarRef guarda el nombre de la variable que se quiere consultar.

```java
package ast;

import java.util.Map;

public class VarRef implements ASTNode {  no usages  Sebas36762
    private String name;  2 usages

    public VarRef(String name) {  1 usage  Sebas36762
        super();
        this.name = name;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {

        return symbolTable.get(name);
    }
```

Mejoras y cambios en el interpreté

Operador lógico AND (&&)

Se creó la regla logic_expr, que permite combinar varios factores mediante el operador AND.

La clase And representa la operación lógica **AND (&&)** en el intérprete. Cada vez que se encuentra una expresión con &&, se crea un nodo And en el árbol de sintaxis abstracta. Este nodo guarda los operandos izquierdo y derecho y, al ejecutarse, evalúa ambos como booleanos y devuelve el resultado de aplicar el operador AND.

```java
package ast;

import java.util.Map;

public class And implements ASTNode {  no usages  Sebas36762
    private ASTNode left;  2 usages
    private ASTNode right;  2 usages

    public And(ASTNode left, ASTNode right) {  2 usages  Sebas36762
        this.left = left;
        this.right = right;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        return (boolean)left.execute(symbolTable) && (boolean)right.execute(symbolTable);
    }
}
```

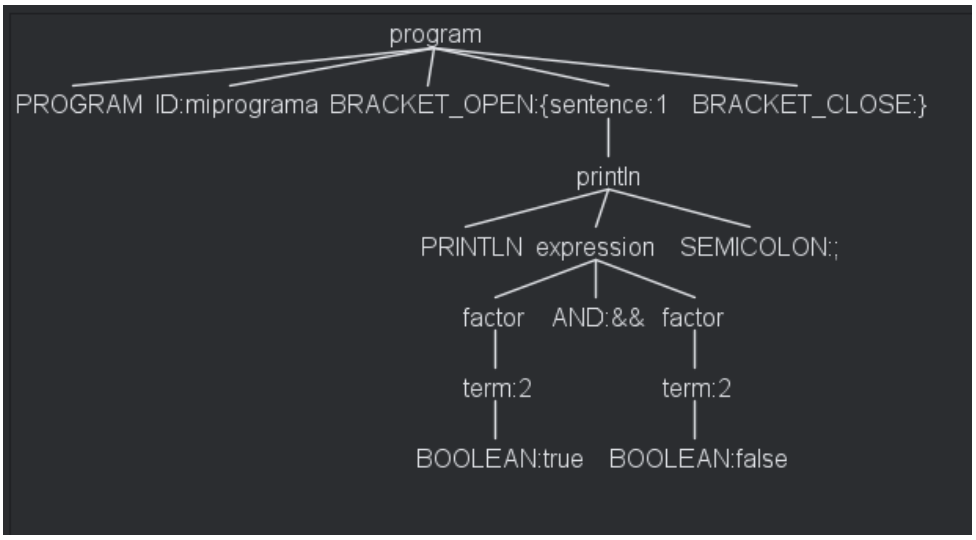
Test realizado:

```
program miprograma {
   println true && false;

}
```

```
Interpetando archivo test/test.smp
false
Interpretacion terminada
```

Esto demuestra que la operación And funciona correctamente.

Arbol de parseo AND:



Fibsum

La clase FibSum representa una operación matemática personalizada que combina la serie de Fibonacci con una multiplicación. Recibe dos operandos: el primero indica cuántos términos de la serie de Fibonacci se suman, y el segundo se multiplica por esa suma.

```java
public class FibSum implements ASTNode {  no usages  Sebas36762
    private ASTNode operand1;  2 usages
    private ASTNode operand2;  2 usages

    public FibSum(ASTNode operand1, ASTNode operand2) {  1 usage  Sebas36762
        this.operand1 = operand1;
        this.operand2 = operand2;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        int x = (int) operand1.execute(symbolTable);
        int y = (int) operand2.execute(symbolTable);

        int sum = 0;
        int a = 1, b = 1;
        for (int i = 1; i <= x; i++) {
            sum += a;
            int temp = a + b;
            a = b;
            b = temp;
        }

        return sum * y;
    }
```

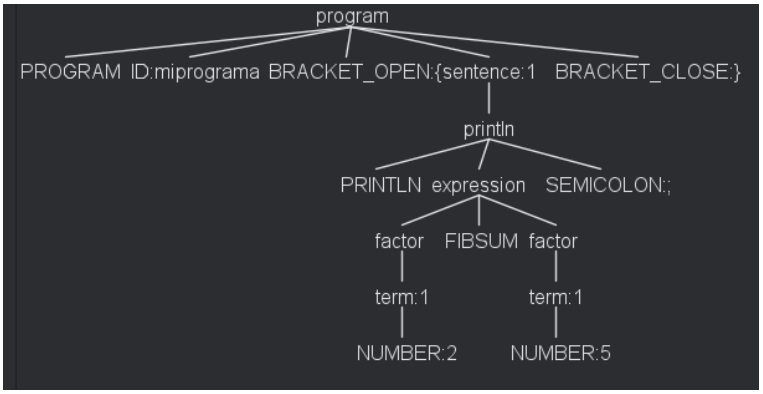
Test Realizado:

```
program miprograma {

    println 5 fibsum 2;
}
```

```
Interpretando archivo test/test.smp
24
Interpretacion terminada
```

Se evalúa la operación 5 fibsum 2. Primero, se suman los primeros 5 términos de la serie de Fibonacci: 1 + 1 + 2 + 3 + 5 = 12. Luego, este resultado se multiplica por 2, dando como resultado final 24.

Arbol de parseo Fibsum:



Pow2Add

La clase Pow2Add representa una operación matemática personalizada que eleva el primer operando al cuadrado y luego le suma el segundo operando.

```java
import java.util.Map;

public class Pow2Add implements ASTNode {  no usages  Sebas36762
    private ASTNode operand1;  2 usages
    private ASTNode operand2;  2 usages

    public Pow2Add(ASTNode operand1, ASTNode operand2) {  1 usage
        this.operand1 = operand1;
        this.operand2 = operand2;
    }

    @Override  Sebas36762
    public Object execute(Map<String, Object> symbolTable) {
        int x = (int) operand1.execute(symbolTable);
        int y = (int) operand2.execute(symbolTable);
        return x * x + y; // x^2 + y
    }
}
```

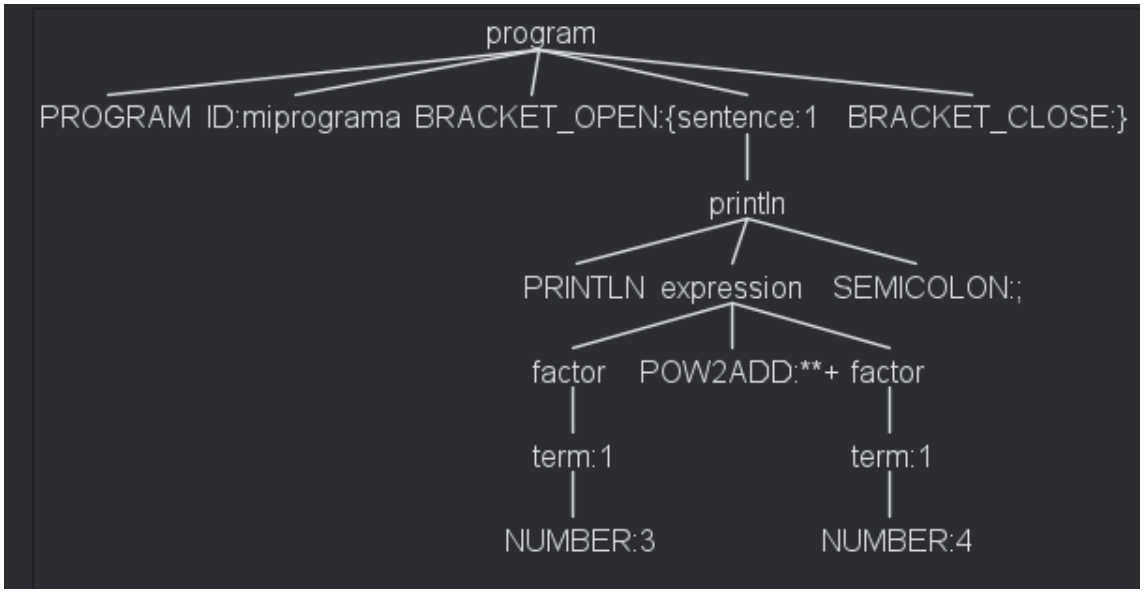
Test Realizado:

```
program miprograma {

    println 3 **+ 4;
}
```

```
Interpretando archivo test/test.smp
13
Interpretacion terminada
```

Se evalúa directamente la operación 3 **+ 4. Primero, el número 3 se eleva al cuadrado, dando 9. Luego, se suma el segundo operando, 4, obteniendo un resultado final de 13.

Arbol de parseo Pow2Add:


program
PROGRAM
ID:miprograma
BRACKET_OPEN:{
sentence:1
BRACKET_CLOSE:}
println
PRINTLN
expression
SEMICOLON:;
factor
POW2ADD:**+
factor
term:1
term:1
NUMBER:3
NUMBER:4

Println

La clase Println se encarga de imprimir en la salida el resultado de una expresión evaluada por el intérprete. Antes de implementar esta clase, al ejecutar varias operaciones seguidas, los resultados se imprimían todos juntos sin separación, por ejemplo: 13false24. Esto dificultaba la lectura y comprensión de la salida.

Ahora cada valor se imprime en una **línea separada**, mejorando significativamente la claridad.

Test Realizado

```
program miprograma {
   var x;
   x = 2;
   println x + 3;
   println true && false;
   println 2 **+ 3;
   println 2 fibsum 5;
   println x;
   println y;
}
```

Resultado antes de la mejora:

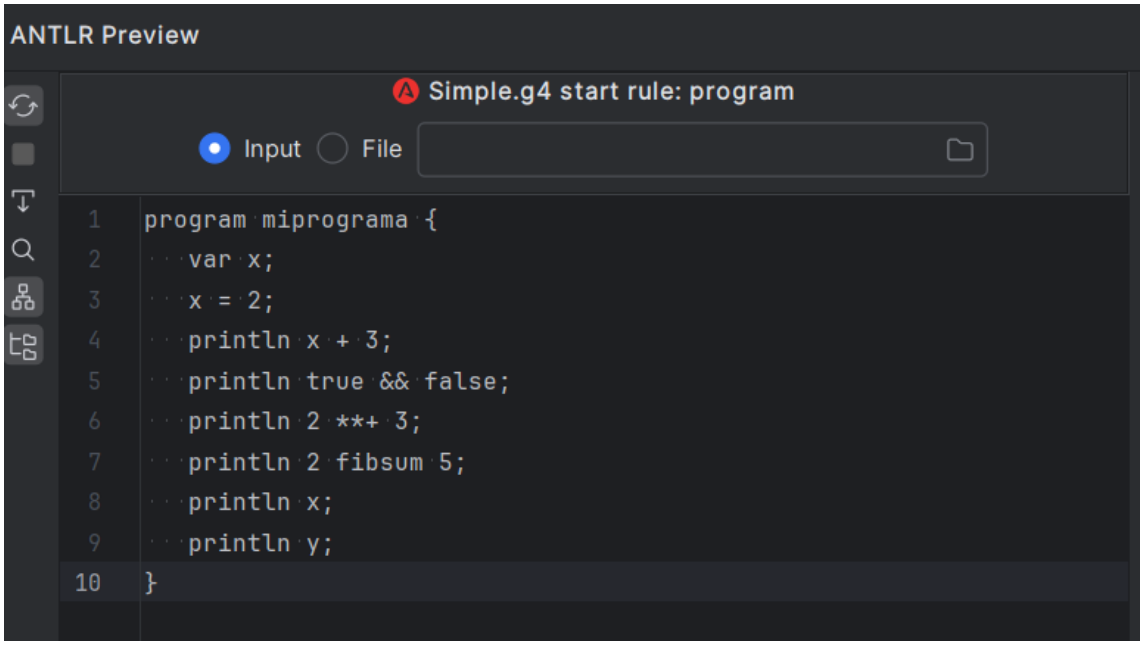
```
Interpretando archivo test/test.smp
5false7102nullInterpretacion terminada

Process finished with exit code 0
```

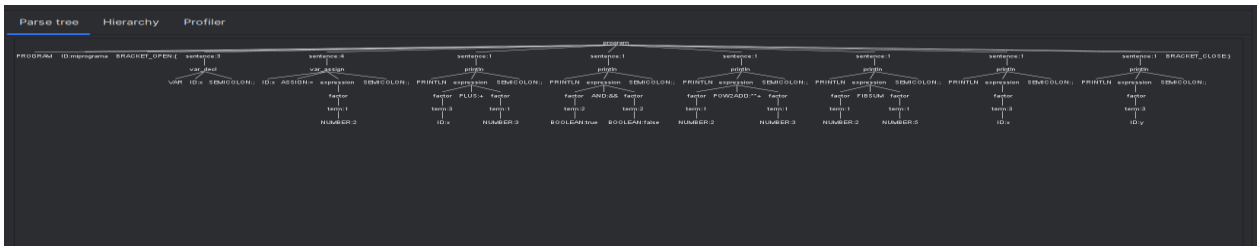
Resultado después de la mejora:

```
Interpretando archivo test/test.smp
5
false
7
10
2
null
Interpretacion terminada
```

Arboles de Sintaxis Abstracta (AST)

Otro ejemplo





Investigación

Conociendo que el análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código, investigar y documentar la forma de utilizar el "análisis semántico" y la "Tabla de Símbolos" para la realización de estas funciones dentro de un compilador.

El análisis semántico utiliza el Árbol de Sintaxis Abstracta (AST), junto con la información recogida en la tabla de símbolos, para verificar que el programa tiene sentido semánticamente. Algunas de las funciones que realiza son:

- Verificar que cada variable utilizada haya sido declarada previamente.
- Verificar tipos compatibles en expresiones (por ejemplo, no sumar un entero con un booleano, si el lenguaje lo prohíbe).

- Verificar reglas de alcance (scope): variables locales, globales, procedimientos o funciones, visibilidad.
- Verificar llamadas a funciones: número de parámetros, tipos, etc.
- En algunos casos, inferencia de tipo o coerción de tipos si el lenguaje lo permite.
- Verificaciones adicionales: que los valores retornados estén correctamente usados, que no haya división por cero (si se modela), que índices de arreglos sean válidos, etc.

La tabla de símbolos es una estructura de datos que almacena información acerca de los símbolos (idents, funciones, clases, constantes, etc) del programa. Algunos de sus contenidos:

- Nombre del símbolo.
- Tipo (int, float, boolean, array, etc).
- Alcance (scope): global, local, bloque, función, módulo, etc.
- Atributos adicionales: tamaño (por ejemplo de arrays), direcciones de memoria asignadas, si es constante o variable, si es mutable, etc.
- Esta información se actualiza durante el análisis semántico y se usa en etapas posteriores.

La tabla de símbolos ayuda al compilador a:

- Resolver identificadores (verificar que se usen correctamente).
- Saber qué tipo tiene cada expresión.
- Saber las propiedades (dirección, tamaño, etc) necesarias para generar código.

Cómo se usan en el backend: generación de código, selección de instrucciones, asignación de registros

Una vez que la fase de análisis semántico ha verificado que el programa está correctamente tipado y bien definido, y la tabla de símbolos tiene la información necesaria, el compilador continúa con el backend. Entre los pasos comunes y típicos, se encuentran:

1. Generación de una representación intermedia (Intermediate Representation, IR):

    a. En ocasiones, la AST se transforma a una IR más conveniente (por ejemplo, “three-address code”, grafos de flujo de control, SSA (Static Single Assignment), etc.).
    b. En este paso se usa la información de tipo y los identificadores almacenados para saber qué operaciones hacer.
2. Optimización de la IR (opcional pero muy habitual):
    a. Eliminación de código muerto.
    b. Propagación de constantes.
    c. Fusión de expresiones comunes.
    d. Optimización de bucles, etc.
    e. Información de la tabla de símbolos puede ayudar para saber qué variables se usan realmente, su alcance, etc.
3. Selección de instrucciones (Instruction Selection): Convierte las operaciones abstractas de la IR en instrucciones concretas de la arquitectura objetivo. Por ejemplo, si la IR establece “multiplica A y B”, elegir “MUL R1, R2” o alguna variante dependiendo del ISA (Instruction Set Architecture). También, aprovecha modos de direccionamiento particulares, instrucciones especiales, etc.
4. Asignación de registros (Register Allocation):
    a. Decidir qué valores o variables estarán en registros de CPU (acceso rápido) y cuáles estarán en memoria.
    b. Elaborar análisis de “vidas vivas” (liveness analysis) para saber cuándo un valor deja de ser necesario, lo que permite reutilizar registros.
    c. Técnicas como coloreo de grafos (graph coloring), “spill” (cuando no hay suficientes registros, mover algunos valores a memoria), etc. Wikipedia+2McGill Ciencia Computación+2
5. Emisión de código final
    a. Después de seleccionar instrucciones y asignar registros, generar el código máquina o el ensamblador para la arquitectura objetivo.
    b. Incluir manejo de llamadas a funciones (prologue / epilogue), acceso a memoria, manejo de pilas, etc.
    c. También generar código para constantes, variables globales, etiquetas, saltos, etc.

Relación entre análisis semántico / tabla de símbolos y backend

Aquí es donde conecta todo:

- La tabla de símbolos proporciona al backend la información de cuales símbolos son los que existen, su tipo, dónde deben almacenarse (memoria, variables locales, globales, registros), el tamaño que van a ocupar, entre otros. Sin esta información, el generador de código no sabría cómo traducir nombres de variables a direcciones o registros.
- El análisis semántico puede enriquecer el AST o la IR con información de tipo, con chequeos, lo cual permite que el backend genere código correcto y optimizado.
- También la semántica se asegura de que los identificadores de las variables en diferentes scopes no choquen, lo que es crucial para generar código que use offsets correctos de pila, o desplazamientos correctos en memoria/stack frame.

Referencias

- Aho, Ullman: "Compilers: Principles, Techniques, and Tools" (El Libro del Dragón)
- Cooper & Torczon: "Engineering a Compiler"
- Appel: "Modern Compiler Implementation"
- *Phases of a Compiler*. (26 de agosto, 2025) https://www.geeksforgeeks.org/compiler-design/phases-of-a-compiler
- *Compiler Design - Quick Guide.* (s.f.). https://www.tutorialspoint.com/compiler_design/compiler_design_quick_guide.htm
- Johnson, M. Zelenski,J. (25 de junio, 2012). *Anatomy of a Compiler.* https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/020%20CS143%20Course%20Overview.pdf
- *Code generation (compiler)*. (s.f.). https://en.wikipedia.org/wiki/Code_generation_%28compiler%29?
- *Análisis semantico: La tabla de símbolos*. (s.f.). https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U5_T2.pdf