Implementación y Mejoras de un Intérprete con ANTLR

Mauricio Luna Acuña
Christian Navarro Ellerbrock
Jorge Gutierrez Vindas
Bryan Stiphen Feng Feng

CE1108 – Compiladores e Intérpretes
I Semestre 2025

17 de septiembre de 2025

Índice

1. Implementación del intérprete

El intérprete se compone de tres etapas: (1) **análisis** con ANTLR (léxico/sintáctico) a partir de la gramática `Simple.g4`; (2) **construcción del AST** mediante acciones semánticas en reglas del parser; y (3) **ejecución** recorriendo el AST con una tabla de símbolos (`Map<String,Object>`) para declaraciones, asignaciones y evaluación de expresiones.

Gramática base y acciones (extracto)

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

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

Interfaz y ejemplo de nodos AST

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

```
public class VarAssign implements ASTNode {
  private final String name; private final ASTNode expr;
  public VarAssign(String name, ASTNode expr) { this.name = name;
     this.expr = expr; }
  public Object execute(Map<String,Object> st) { st.put(name, expr.
     execute(st)); return null; }
}
```

Ejemplo de uso:

```
program miprograma {
  var x; x = 5;
  if (false) { println 2 + 2 * 4; } else { println x; } // imprime
     5
}
```

2. Mejoras realizadas en la gramática y análisis semántico

2.1. Operación matemática lógica (igualdad ==)

Se incorporó el operador lógico de igualdad con precedencia *menor* que + y *. La regla superior combina `addExpr` con posibles comparaciones encadenadas:

```
expression returns [ASTNode node]
  : a=addExpr                { $node = $a.node; }
    ( EQ b=addExpr           { $node = new Equal($node, $b.node); }
       )*
  ;

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

Nodo `Equal` (compara números y booleanos):

```
public class Equal implements ASTNode {
  private final ASTNode left, right;
  public Equal(ASTNode l, ASTNode r){ this.left=l; this.right=r; }
  public Object execute(Map<String,Object> st) {
    Object a = left.execute(st), b = right.execute(st);
    if (a==null || b==null) return a==b;
    if (a instanceof Number && b instanceof Number)
      return ((Number)a).intValue() == ((Number)b).intValue();
    if (a instanceof Boolean && b instanceof Boolean)
      return ((Boolean)a).booleanValue() == ((Boolean)b).
         booleanValue();
    return a.equals(b);
  }
}
```

Ejemplo:

```
println 2 + 2 * 4 == 10; // true
```

2.2. Operación matemática creativa

Se definió un nuevo operador personalizado `@`, llamado **sumaplicación**, el cual combina suma y multiplicación. Su semántica es:

$$a \,@\, b = a + b + (a \times b)$$

Este operador fue incorporado al mismo nivel de precedencia que la suma (+) en la gramática.

Listing 1: Regla para @ en la gramática

```
addExpr returns [ASTNode node]
  : t1=multExpr { $node = $t1.node; }
    ( PLUS t2=multExpr { $node = new Addition($node, $t2.node); }
    | AT   t3=multExpr { $node = new Sumaplicacion($node, $t3.node); } )
       *;
```

Listing 2: Nodo Sumaplicacion en el AST

```
public class Sumaplicacion implements ASTNode {
    private final ASTNode left, right;

    public Sumaplicacion(ASTNode left, ASTNode right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public Object execute(Map<String, Object> symbolTable) {
        int a = (int) left.execute(symbolTable);
        int b = (int) right.execute(symbolTable);
        return a + b + (a * b);
    }
}
```

Ejemplo:

```
println 2 @ 3;    // Resultado: 2 + 3 + (2*3) = 11
println 1 @ 0;    // Resultado: 1 + 0 + (1*0) = 1
```

2.3. Operación probabilística (permutaciones `nPr` con `P`)

Se incorporó el operador de permutación $nP_r = \frac{n!}{(n-r)!} = \prod_{k=n-r+1}^{n} k$ con la **misma precedencia** que `*`. Para evitar conflictos léxicos ("*maximal munch*"), se usa **espacio** alrededor de `P` (p. ej., `3 P 2`).

```
multExpr returns [ASTNode node]
  : t1=term { $node = $t1.node; }
    ( MULT t2=term { $node = new Multiplication($node, $t2.node); }
    | PERM t3=term { $node = new Permutation($node, $t3.node); } )*
  ;
PERM: 'P';
```

Nodo `Permutation` (cálculo como producto, con validaciones):

```
public class Permutation implements ASTNode {
  private final ASTNode nNode, rNode;
  public Permutation(ASTNode n, ASTNode r){ this.nNode=n; this.
     rNode=r; }
  public Object execute(Map<String,Object> st) {
    int n=((Number)nNode.execute(st)).intValue();
    int r=((Number)rNode.execute(st)).intValue();
    if(n<0||r<0) throw new IllegalArgumentException("n,r␣>=␣0");
    if(r>n) return 0;
    long res=1L; for(int k=n-r+1;k<=n;k++){ res*=k;
```

```
        if(res>Integer.MAX_VALUE) throw new ArithmeticException("
            overflow 32-bit"); }
      return (int)res;
    }
}
```

Ejemplos:

```
println 3 P 2;      // 6
println 5 P 3;      // 60
var n; var r; n=6; r=2; println n P r; // 30
```

Nota léxica. Si se desea admitir `3P2` sin espacios, el lexer debe definir tokens compuestos antes de `ID`/`NUMBER` (p. ej., `NPR_NUM: [0-9]+ 'P' [0-9]+;`) y el parser debe construir el nodo `Permutation` a partir de dicho token; en este informe optamos por la forma con espacio para mantener la gramática simple y la precedencia en el nivel multiplicativo.

3. Árbol de parseo y resultados

En esta sección se presentan los árboles de parseo generados por ANTLR, junto con los programas ejecutados y sus respectivos resultados en el intérprete. Cada imagen fue capturada desde la vista de ANTLR Parse Tree Viewer en Eclipse.

3.1. Árbol de parseo básico del programa

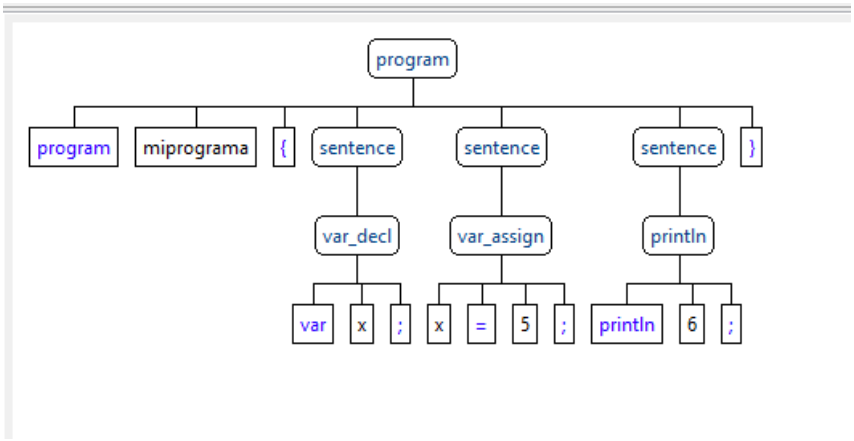

Figura 1: Árbol de parseo: estructura base de un programa (program ID { oraciones })

Listing 3: Código ejecutado

```
program miprograma {
  var x;
  x=5;
  println 6;
}
```

Salida esperada: No se imprime nada. Solo se declara la variable.

3.2. Árbol: suma y multiplicación sin paréntesis

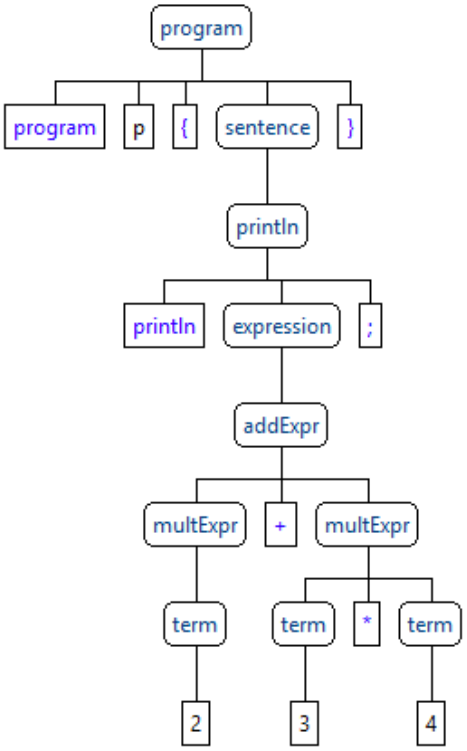

Figura 2: Árbol de parseo: 2 + 3 * 4 (la multiplicación se evalúa antes)

Listing 4: Código ejecutado

```
program p {
  println 2 + 3 * 4;
}
```

Salida: `14`

3.3. Árbol: suma con paréntesis

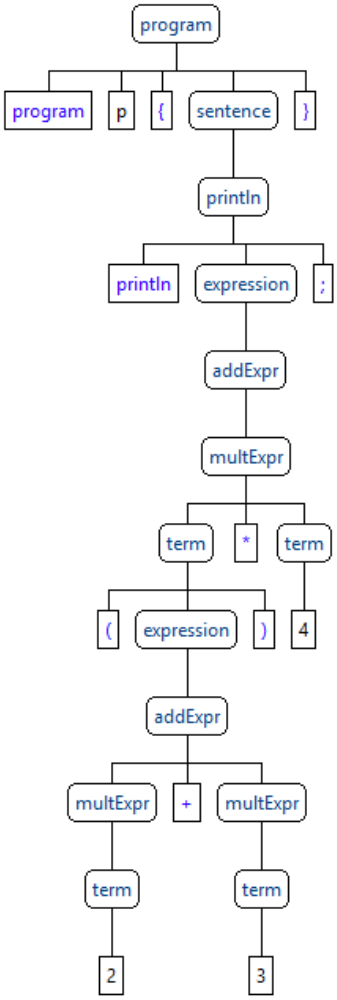

Figura 3: Árbol de parseo: (2 + 3) * 4 (la suma ocurre primero)

Listing 5: Código ejecutado

```
program p {
  println (2 + 3) * 4;
}
```

Salida: 20

3.4. Árbol: estructura condicional if / else

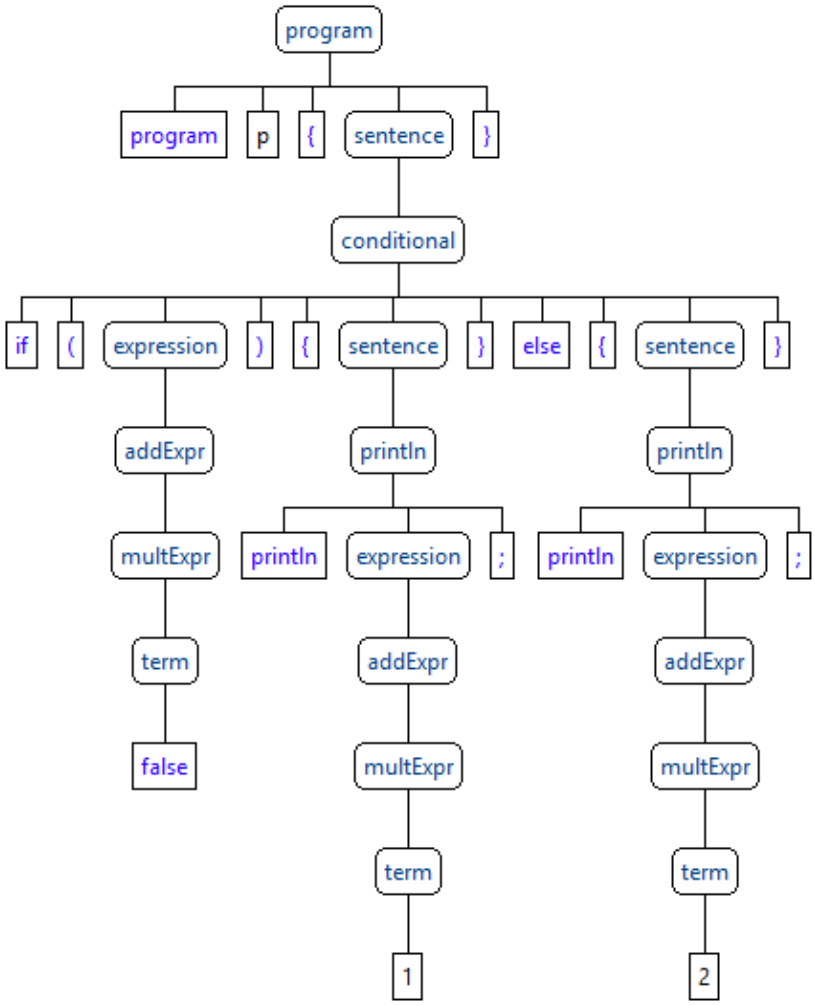

Figura 4: Árbol de parseo: condicional if/else básico

Listing 6: Código ejecutado

```
program p {
  if (false) {
    println 1;
  } else {
    println 2;
  }
}
```

Salida: 2

3.5. Árbol: operación de igualdad (==)

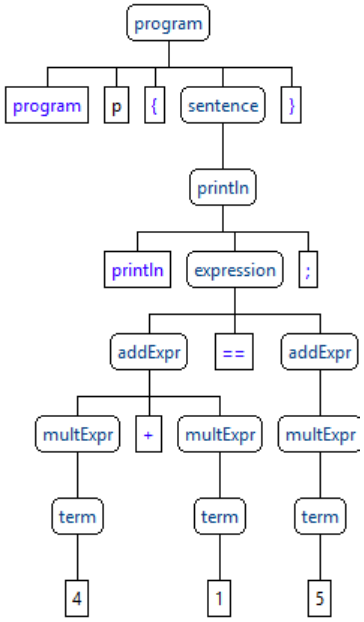

Figura 5: Árbol de parseo: comparación con ==

Listing 7: Código ejecutado

```
program p {
  println 4 + 1 == 5;
}
```

Salida: `true`

3.6. Árbol: operación sumaplicación (@)

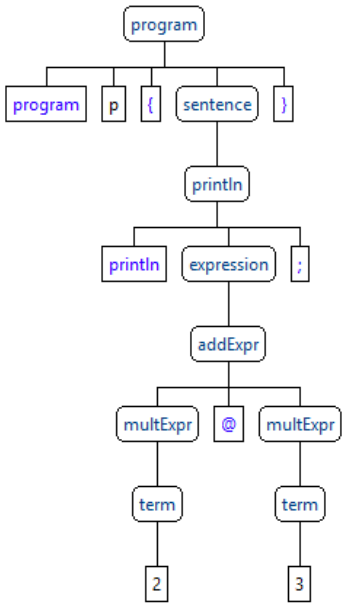

Figura 6: Árbol de parseo: operador creativo @

Listing 8: Código ejecutado

```
program p {
  println 2 @ 3;
}
```

Salida: `11`

3.7. Árbol: operación probabilística (P)

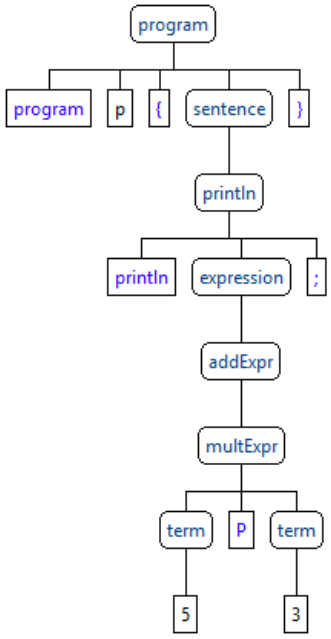

Figura 7: Árbol de parseo: operador de permutación n P r

Listing 9: Código ejecutado

```
program p {
  println 5 P 3;
}
```

Salida: `60`

4. Investigación: Análisis semántico y Tabla de Símbolos

El análisis semántico es una fase fundamental dentro del proceso de compilación o interpretación. Se ejecuta después del análisis sintáctico y recibe como entrada el árbol de parseo o AST (Árbol de Sintaxis Abstracta). Su objetivo es verificar que el programa tenga sentido más allá de la forma: comprobar tipos, existencia de variables, número correcto de parámetros en llamadas, entre otros.

4.1. Objetivos del análisis semántico

- Verificar que las operaciones sean válidas según el tipo de datos (por ejemplo, no sumar una cadena con un entero si el lenguaje no lo permite).
- Comprobar que todas las variables y funciones usadas hayan sido previamente declaradas.
- Validar reglas de alcance (scope) y visibilidad de identificadores.

- Preparar la información necesaria para la generación de código o para la ejecución directa.

4.2. Tabla de símbolos

La **tabla de símbolos** es una de las estructuras de datos más importantes en el proceso de compilación e interpretación. Actúa como un *diccionario del programa*, donde se almacenan y gestionan todos los identificadores declarados: variables, constantes, funciones, procedimientos, parámetros e incluso símbolos temporales generados por el compilador o intérprete.

¿Qué información almacena?

Cada entrada en la tabla de símbolos suele incluir:

- **Nombre**: el identificador tal como aparece en el código fuente.
- **Tipo**: entero, real, cadena, booleano, etc.
- **Ámbito o nivel**: en qué bloque del programa fue declarado y su visibilidad.
- **Dirección o referencia**: posición en memoria o en una estructura interna.
- **Información adicional**: parámetros de funciones, si es constante o variable, valor inicial, etc.

Ejemplo práctico

Si el programa contiene:

```
var x: integer;
var y: real;
function suma(a: integer; b: integer): integer;
```

La tabla de símbolos podría representarse así:

Identificador	Tipo	Ámbito	Información
x	integer	global	variable
y	real	global	variable
suma	función	global	parámetros: (a: integer, b: integer); retorno: integer
a	integer	local (suma)	parámetro por valor
b	integer	local (suma)	parámetro por valor

Operaciones básicas

La tabla de símbolos debe permitir tres operaciones fundamentales:

- **Insertar**: agregar nuevas declaraciones (ej. al encontrar `var x: integer;`).
- **Buscar**: consultar si un símbolo ya existe y recuperar su información (ej. al evaluar una expresión con `x + 1`).
- **Eliminar**: borrar o desactivar símbolos al cerrar un bloque o función (reglas de alcance).

Aplicación en este proyecto

En el intérprete desarrollado con ANTLR:

- La tabla de símbolos se usa para

5. Investigación: Análisis semántico y Tabla de Símbolos

El **análisis semántico** es una fase crucial en el proceso de compilación o interpretación. Se ejecuta después del análisis sintáctico y utiliza el Árbol de Sintaxis Abstracta (AST) como entrada. Su objetivo principal es verificar que el programa no solo sea correcto en su forma (sintaxis), sino también en su significado. Es decir, se asegura de que las operaciones, las declaraciones y el uso de identificadores respeten las reglas del lenguaje de programación.

5.1. Objetivos del análisis semántico

Entre los principales objetivos de esta fase se encuentran:

- Verificar la **compatibilidad de tipos** en operaciones (por ejemplo, no sumar un entero con una cadena si el lenguaje no lo permite).
- Comprobar que todas las variables y funciones **hayan sido declaradas** antes de usarse.
- Validar las **reglas de alcance y visibilidad**, garantizando que se use la declaración más cercana en caso de identificadores repetidos en distintos bloques.
- Comprobar el número y tipo de **parámetros** en las llamadas a funciones o procedimientos.
- Preparar información para fases posteriores, como la generación de código o la ejecución en un intérprete.

5.2. Tabla de símbolos

La **tabla de símbolos** es la estructura de datos fundamental que soporta al análisis semántico. Funciona como un *diccionario del programa*, donde cada identificador declarado queda registrado junto con la información necesaria para su verificación y posterior uso.

Contenido típico

Cada entrada de la tabla de símbolos suele almacenar:

- Nombre del identificador.
- Tipo de dato (entero, real, cadena, booleano, etc.).
- Nivel de alcance o bloque en el que fue declarado.

- Dirección de memoria o referencia en estructuras internas.
- Información adicional: parámetros de funciones, valor inicial, si es constante o variable, etc.

Operaciones principales

Las operaciones esenciales sobre la tabla de símbolos son:

- **Insertar**: registrar nuevas declaraciones.
- **Buscar**: consultar información de un identificador al ser utilizado en el programa.
- **Eliminar**: retirar símbolos cuando se cierra un bloque o ámbito.

Ejemplo ilustrativo

Dado el siguiente fragmento en un pseudolenguaje:

```
var x: integer;
var y: real;
function suma(a: integer; b: integer): integer;
```

La tabla de símbolos podría verse así:

Identificador	Tipo	Ámbito	Información
x	integer	global	variable
y	real	global	variable
suma	función	global	parámetros: (a: integer, b: integer); retorno: integer
a	integer	local (suma)	parámetro por valor
b	integer	local (suma)	parámetro por valor

5.3. Aplicación al proyecto

En el intérprete desarrollado con ANTLR, esta investigación se aplica de la siguiente forma:

- La tabla de símbolos mantiene un registro de las variables y sus valores en tiempo de ejecución.
- El análisis semántico garantiza que las variables estén declaradas antes de usarse y que las operaciones respeten los tipos.
- Las operaciones nuevas (lógica, creativa y probabilística) se apoyan en la tabla de símbolos para validar que los operandos son adecuados.
- Gracias a esta verificación, se pueden detectar errores semánticos de forma temprana, mejorando la robustez del intérprete.

Referencias

[1] Jaime Pavlich. *Tutoriales ANTLR para construir un intérprete*. Canal de YouTube: `https://www.youtube.com/@jaimepavlich` Consultado en septiembre de 2025.

[2] J. Neira. *Compiladores II – Tabla de Símbolos*. Universidad de Zaragoza, 12048 Compiladores II. Disponible en línea (material del curso, acceso por docente). Consultado en septiembre de 2025.