

PRODUCTO INTEGRADOR: PORTAFOLIO HUGGING FACE

Integrantes:

Hernandez Balleza José Eduardo

Moctezuma Herrera Katya Paola

Torres Hipolito Carlos Manuel

PROGRAMACIÓN DE SISTEMA DE BASE 2

Muñoz Quintero Dante Adolfo

9G

Especificación del Proyecto Final: Traductor con Integración de Hugging Face

Materia: Programación de Sistemas de Base 2

Duración: Semestral

Objetivo

Desarrollar un traductor que integre modelos de Hugging Face para potenciar el software (p. ej. análisis semántico y generación de código intermedio). El proyecto podrá utiliza NLP para mejorar los mensajes de error, sugerir correcciones u optimizar el código intermedio basado en patrones aprendidos.

Alcance

- Lenguaje Personalizado:** Definir un lenguaje con sintaxis y reglas semánticas específicas (ej: un DSL para cálculos matemáticos o manipulación de datos).
 - Análisis Semántico:** Verificación de tipos, manejo de tablas de símbolos y detección de errores contextuales.
 - Generación de Código Intermedio:** Producción de representaciones como código de tres direcciones o una estructura intermedia optimizable.
 - Integración con Hugging Face:**
 - Usar modelos preentrenados (ej: CodeBERT, GPT-2) para generar mensajes de error explicativos o sugerencias de corrección.
 - Implementar un módulo que convierta comentarios en lenguaje natural en anotaciones estructuradas para el código intermedio.
-

Detalles del Proyecto

- Herramientas:**
 - ANTLR o Lex/Yacc para análisis léxico/sintáctico.
 - Hugging Face Transformers o Inference API para integrar modelos de NLP.
 - Python o Java como lenguaje de implementación.
- Ejemplo de Funcionalidad:**

- Si un usuario escribe `int x = "texto";`, el compilador detecta un error de tipo y usa un modelo de Hugging Face para sugerir: “¿Quisiste asignar un valor numérico o cambiar el tipo de variable a ‘string’?”.
 - Traducir comentarios como `// sumar los elementos del arreglo` en anotaciones para optimizar bucles en el código intermedio.
-

Entregables

1. Archivos Generados

- Código fuente del compilador (ej: gramáticas, analizadores semánticos, generadores de código).
- Ejemplos de entrada/salida:

- **Entrada:**

```
var x: int = "hola"; # Error de tipo
```

- **Salida:**

```
Error en línea 1: Tipo incompatible. Sugerencia (Hugging Face): ¿Intentaste asignar un 'string' a una variable de tipo 'int'?
```

2. Documentación Técnica

- **Descripción del Lenguaje:** Sintaxis BNF, reglas semánticas y casos de uso.
- **Decisiones de Diseño:**
 - Elección de modelos de Hugging Face (ej: CodeBERT para contexto de código).
 - Optimización del código intermedio usando anotaciones generadas por NLP.

3. Informe Final

- **Introducción:** Relevancia de integrar NLP en compiladores.
- **Diagramas:**
 - Árbol sintáctico con anotaciones semánticas.
 - Flujograma del proceso de compilación con interacción de Hugging Face.
- **Pruebas:** Casos de éxito, manejo de errores y evaluación del impacto de los modelos en la experiencia del usuario.

4. Consideraciones Finales

- Los equipos pueden ajustar el alcance según su complejidad, pero deben justificar cualquier desviación.
 - Se recomienda utilizar herramientas y lenguajes conocidos (ejemplo: Python, Java, C++) para agilizar el desarrollo.
 - La claridad en la definición de tokens y gramática es crítica para el éxito del proyecto.
-

Repositorio y Readme

Se deberá entregar todos el software relativo al proyecto en un enlace a un repositorio personal público y accesible.

Posible Estructura del Repositorio:

```
📁 Proyecto_Compiladores2/
├── 📄 compiler/           # Código fuente
├── 📄 ejemplos/          # Casos de prueba
├── 📄 docs/              # Documentación técnica
├── 📄 informe_final.pdf
└── 📄 README.md
```

Contenido del Readme:

```
# Compilador con NLP para Mensajes Inteligentes
Compilador que integra Hugging Face para mejorar mensajes de error y código intermedio.

## Información del Curso
* Materia: Compiladores 2
* Institución: Universidad XYZ
* Semestre: Segundo semestre de 2023
* Profesor: Dr. Juan Pérez

## Integrantes
```

- María García
 - Luis Rodríguez
-

Requisitos Adicionales

- Subir a Moodle un PDF con código fuente y capturas de:
 - Ejecución exitosa con mensajes de Hugging Face.
 - Errores manejados por el modelo (ej: pantalla de sugerencia de tipo).
-

Integridad Académica

El código debe ser original. Se permiten referencias externas, pero deben citarse. Cualquier plagio resultará en calificación cero.

Este proyecto busca explorar cómo la IA puede hacer los compiladores más accesibles y eficientes, combinando teoría de compilación con herramientas modernas de NLP.

Objetivo

Desarrollar un analizador sintáctico utilizando ANTLR para procesar un lenguaje personalizado, asegurando una tokenización eficiente que facilite análisis semántico posterior.

Introducción

En el presente documento se expone una nueva propuesta de lenguaje de programación denominada “ProLang”. Con este propósito, se presenta al lector los tokens, las gramáticas que rigen el lenguaje, y una serie de ejemplos diseñados para facilitar la comprensión de su sintaxis.

En este contexto, la implementación de lenguaje ProLang se logró hasta las primeras dos fases de la compilación: análisis léxico y análisis sintáctico. Esto se logró gracias a ANTLR, una herramienta poderosa para la construcción de analizadores léxicos y sintácticos.

Enlace al repositorio del proyecto

https://github.com/x-vills/Proyecto_Compiladores2/

Alcance

El lenguaje de programación ProLang

ProLang es un lenguaje de programación diseñado con fines académicos que permite expresar construcciones comunes en el paradigma de programación estructurada mediante una sintaxis clara y no tan completa como otros lenguajes de programación.

Un programa escrito en ProLang inicia y finaliza con las palabras clave start y end, respectivamente, las cuales proporcionan un marco bien definido para el cuerpo del código.

Admite declaraciones de variables utilizando la instrucción `define`, la cual permite especificar o definir el nombre de un identificador o variable, su tipo de dato y, opcionalmente, una expresión de inicialización.

El lenguaje incluye gramáticas para estructuras de control condicionales y cíclicas, y una función de impresión a través de la instrucción `print`, que admite uno o varios argumentos.

Enfoque de ProLang

Dara apoyo a la educación ya que facilitara el estudio de los lenguajes de programación y el funcionamiento de los analizadores sintácticos con su enfoque en compiladores.

ProLang

Sintaxis:

```
1 start
2   define @x as integer = 10;
3   define @y as decimal = 5.5;
4   define @z as boolean = true;
5   define @name as varchar = "John";
6   define @result as generic;
7
8   print(@x, @y, @z, @name);
9
10  @x = (@x + 5) * 2;
11  @z = !false;
12
13  if (@x > 10) :
14    print("x is greater than 10");
15    if (@z == true) :
16      print("z is true");
17    else (@z == false) :
18      print("z is false");
19    end
20  else :
21    print("x is not greater than 10");
22  end
23
24  switch (@x) :
25    case 10:
26      print("Value is 10");
27      break;
28    case 20:
29      print("Value is 20");
30      break;
31    default:
32      print("Value is something else");
33  end
34
35  while (@x < 25) :
36    print("In while:", @x);
37    @x = @x + 1;
38  end
39
40  for (define @i as integer = 0; @i < 5; @i = @i + 1) :
41    print("For loop i:", @i);
42  end
43 end
```

Reglas semánticas

Evaluación de Expresiones

Las expresiones deben respetar las prioridades matemáticas y de operadores.

- Ejemplo válido:
- define @x as integer = ((5 + 3) * 2) / 4;
- Ejemplo inválido:
- define @y as boolean = 5 + true; // Error: Tipos incompatibles

Declaración y Uso de Variables

Una variable debe declararse antes de su uso. Se debe respetar el tipo de dato asignado en la declaración.

- Ejemplo válido:
- `define @x as integer = 10;`
- `print(@x); // Correcto`
- Ejemplo inválido:
- `print(@y); // Error: @y no ha sido declarado`

Restricciones de Tipos de Datos

Operaciones matemáticas solo son válidas con ``integer`` y ``decimal``. Operaciones lógicas requieren operandos ``boolean``.

- Ejemplo válido:
- `define @flag as boolean = (true && false);`
- Ejemplo inválido:
- `define @result as integer = "Texto" + 5; // Error: Operación inválida`

Asignación de Tipos Consistentes

No se puede asignar un ``varchar`` a una variable ``integer``. No se puede asignar ``boolean`` a ``decimal`` o ``integer``.

- Ejemplo inválido:
- `define @x as integer = "Hola"; // Error: Tipo incompatible`

Compatibilidad de Comparaciones

Comparaciones deben realizarse entre tipos compatibles.

- Ejemplo válido:
- `if (@x > 10) : print("X es mayor a 10"); end`
- Ejemplo inválido:
- `if (@x > "Hola") : print("X es mayor a Hola"); end // Error: Comparación no válida`

Corrección de Estructuras de Control

- Un bloque ``if``, ``switch``, ``while`` o ``for`` debe cerrarse con ``end``.
- Ejemplo válido:
- `if (@x > 10) :`
- `print("X es mayor a 10");`
- `end`
- Ejemplo inválido:
- `if (@x > 10) :`
- `print("X es mayor a 10"); // Error: Falta 'end'`

Análisis Semántico

El análisis semántico no se implementó, pero es importante considerar lo siguiente para aprovechar la herramienta:

- **Verificación de tipos:** garantiza que las variables y las expresiones respeten reglas de lenguaje estos son los tipos de datos.
 - integer → Números enteros (10, -5)
 - decimal → Números flotantes (3.14, 0.99)
 - boolean → Valores lógicos (true, false)
 - varchar → Texto ("hello", "data")
 - generic → Tipo indefinido hasta ejecución
- **Ejemplo de verificación de tipos**

Código valido

```
define @x as integer = 10;  
  
define @y as varchar = "Hola";
```

Código con error de tipo:

```
define @x as integer = "Hola"; // Error: Se esperaba un 'integer'
```

- **Manejo de tabla de símbolos:** Esta se usa para almacenar la información de cada identificador incluyendo:
 - Nombre de la variable (@x, @y)
 - Tipo de dato (integer, decimal, etc.)
 - Valor actual (si está inicializado)
 - Ámbito (global o dentro de una función/ciclo)
- **Ejemplos:**
 - define @x as integer = 10;

- define @y as boolean = false;

- **La tabla de símbolos se vería así:**

@x	Integer	10	Global
@y	Boolean	False	global

- **Manejo de errores contextuales:** Estos errores ocurrirán cuando el código es léxicamente incorrecto o ignora las reglas semánticas del mismo lenguaje, algunos ejemplos son:
 - Uso de variables no declaradas
 - Operaciones inválidas entre tipos diferentes
 - Estructuras mal formadas (if sin end)

Integración con ANTLR

Se utilizó la función de este mismo para poder generar el analizador léxico y sintáctico ya que ofrece una mejor compatibilidad y evita errores en la programación. También se configuró para la muestra de los árboles sintácticos.

Detalles del proyecto

Herramientas:

- Se utilizó ANTLR para generar el analizador sintáctico.

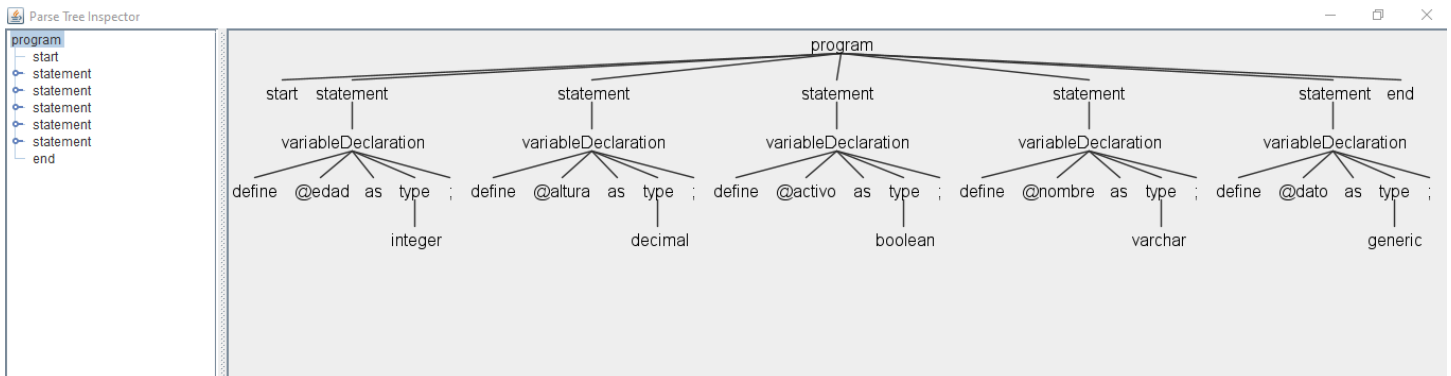
Ejemplos de Funcionalidad.

Declaración de variables sin asignación

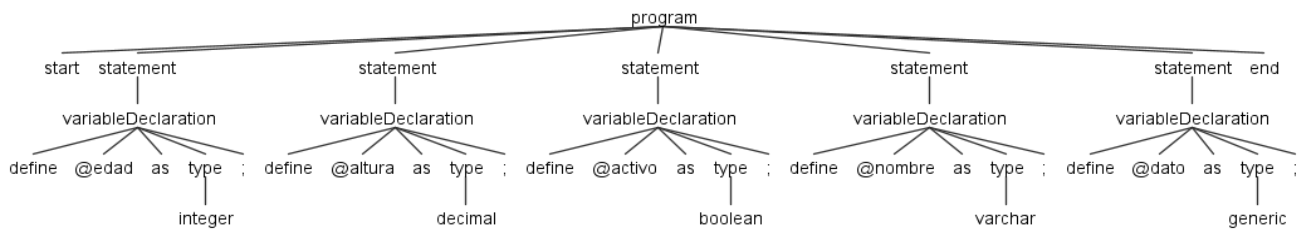
- Entrada

```
PS C:\Users\carlo\OneDrive\Escritorio\Parser> grun ProLang program -gui
start
  define @edad as integer;
  define @altura as decimal;
  define @activo as boolean;
  define @nombre as varchar;
  define @dato as generic;
end
^Z
```

- Salida



○ Árbol



Función print multiparámetro

○ Entrada

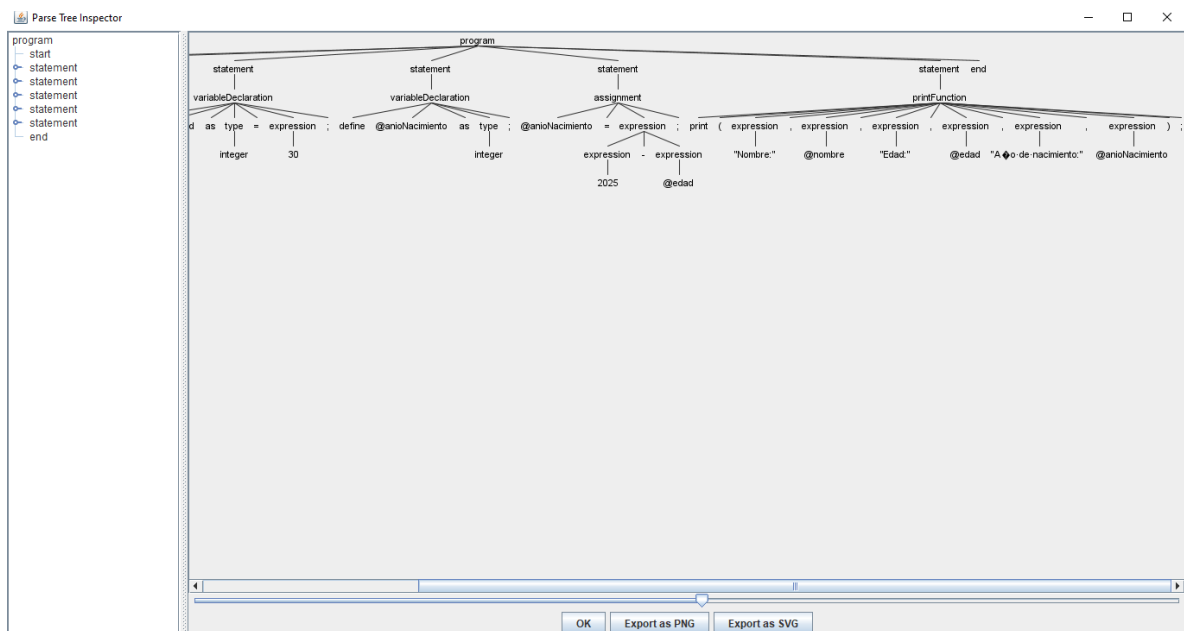
```

PS C:\Users\carlo\OneDrive\Escritorio\Parser> grun ProLang program -gui
start
  define @nombre as varchar = "Ana";
  define @edad as integer = 30;
  define @anioNacimiento as integer;

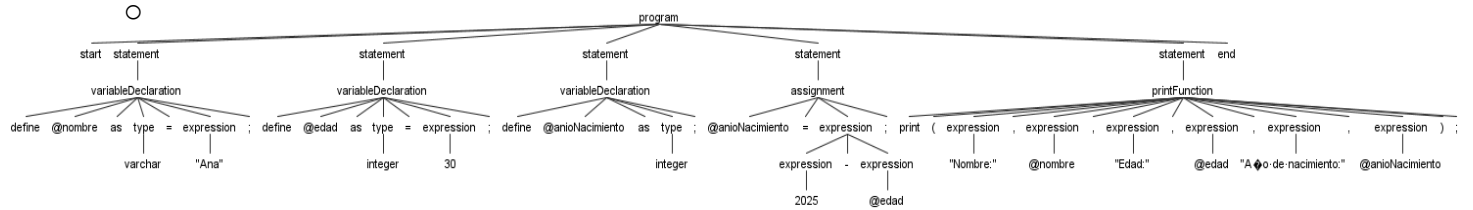
  @anioNacimiento = 2025 - @edad;

  print("Nombre:", @nombre, "Edad:", @edad, "Año de nacimiento:", @anioNacimiento);
end
^Z
_
  
```

○ Salida



- Árbol



Estructura cíclica While

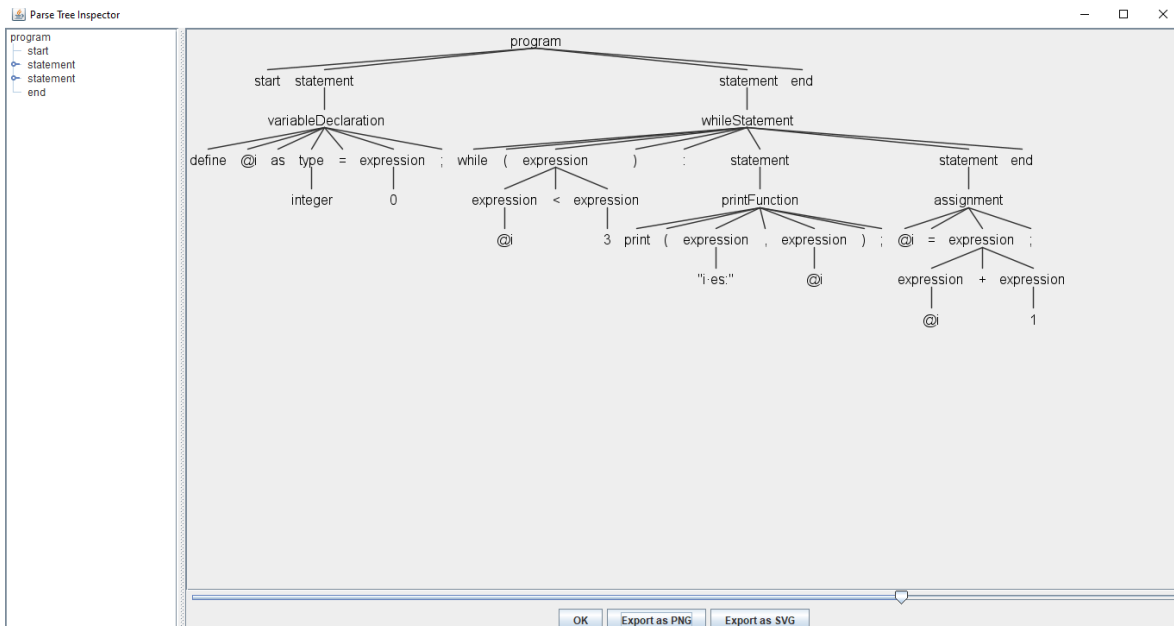
- Entrada

```

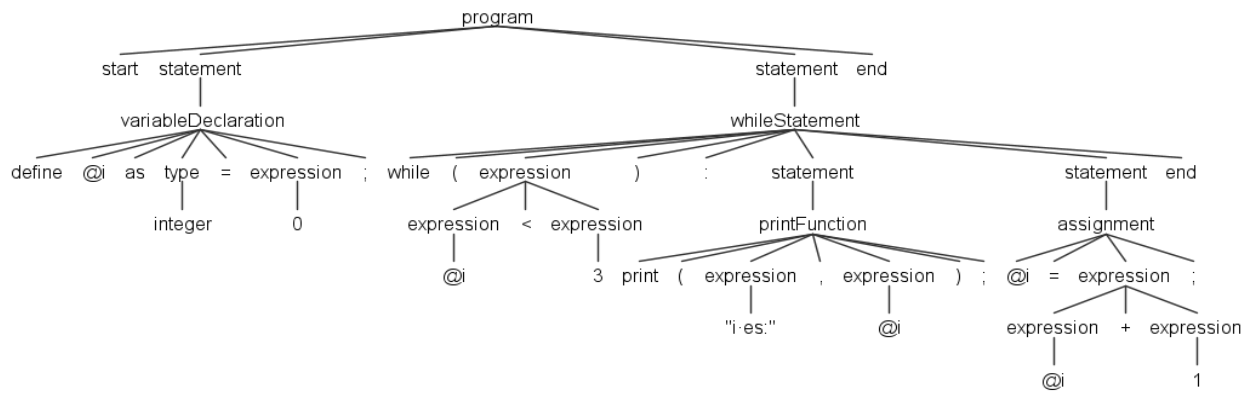
PS C:\Users\carlo\OneDrive\Escritorio\Parser> grun ProLang program -gui
start
  define @i as integer = 0;

  while (@i < 3):
    print("i es:", @i);
    @i = @i + 1;
  end
end
^Z
  
```

- Salida



Arbol



Por último, colocaremos el código fuente y captura donde se implementó.

El único archivo fuente que se generó manualmente fue ProLang.g4, los demás se omiten dado que fueron generados por ANTLR.

```
grammar ProLang;

// Gramáticas
program
    : START statement* END
    ;

statement
    : variableDeclaration
    | assignment
    | ifStatement
    | switchStatement
    | whileStatement
    | forStatement
    | printFunction
    ;

variableDeclaration
    : DEFINE ID AS type (ASSIGN expression)? SEMICOLON
    ;

printFunction
    : PRINT LPAREN (expression (',' expression)*) RPAREN SEMICOLON
    ;

assignment
    : ID ASSIGN expression SEMICOLON
    ;

ifStatement
    : IF LPAREN expression RPAREN COLON statement*
      (ELSE LPAREN expression RPAREN COLON statement*)*
      (ELSE COLON statement*)?
      END
    ;

switchStatement
    : SWITCH LPAREN expression RPAREN COLON
      caseBlock+
      (DEFAULT COLON statement*)?
      (DEFAULT COLON statement*)?
      END
    ;

caseBlock
    : CASE expression COLON statement* BREAK SEMICOLON
    ;

whileStatement
    : WHILE LPAREN expression RPAREN COLON statement* END
    ;

forStatement
    : FOR LPAREN forInit SEMICOLON forCondition SEMICOLON forUpdate RPAREN COLON statement* END
    ;
```



```

forStatement
    : FOR LPAREN forInit SEMICOLON forCondition SEMICOLON forUpdate RPAREN COLON statement* END
    ;

```

```

forInit
    : DEFINE ID AS INTEGER ASSIGN expression
    | ID ASSIGN expression
    ;

```

```

forCondition
    : expression
    ;

```

```

forUpdate
    : ID ASSIGN expression
    ;

```

```

type
    : INTEGER
    | DECIMAL
    | BOOLEAN
    | VARCHAR
    | GENERIC
    ;

```

```

expression
    : LPAREN expression RPAREN #parenExpression
    | left=expression op=POWER right=expression #binaryOp
    | (NOT | MINUS) expression #unaryOp
    | left=expression op=(MULTIPLICATION | DIVISION | MOD) right=expression #binaryOp
    | left=expression op=(PLUS | MINUS) right=expression #binaryOp
    | left=expression op=(GREATER_THAN | GREATER_OR_EQUAL | LESS_THAN | LESS_OR_EQUAL) right=expression #binaryOp
    | left=expression op=(EQUAL_THAN | DIFFERENT_THAN) right=expression #binaryOp
    | left=expression AND right=expression #binaryOp
    | left=expression OR right=expression #binaryOp
    | ID #idExpression
    | NUMBER #numberExpression
    | TEXT #textExpression
    | boolean #booleanExpression
    ;

```

```

boolean
    : TRUE
    | FALSE
    ;

```

```

NUMBER
    : INTEGER_NUMBER
    | DECIMAL_NUMBER
    ;

```

```

// Operadores relacionales
AND : 'and' | '&&';
OR  : 'or' | '||';
NOT : '!';

```

```

// Operadores matemáticos
PLUS : '+';
MINUS : '-';
MULTIPLICATION : '*';
DIVISION : '/';
MOD : '%';
POWER : '^';

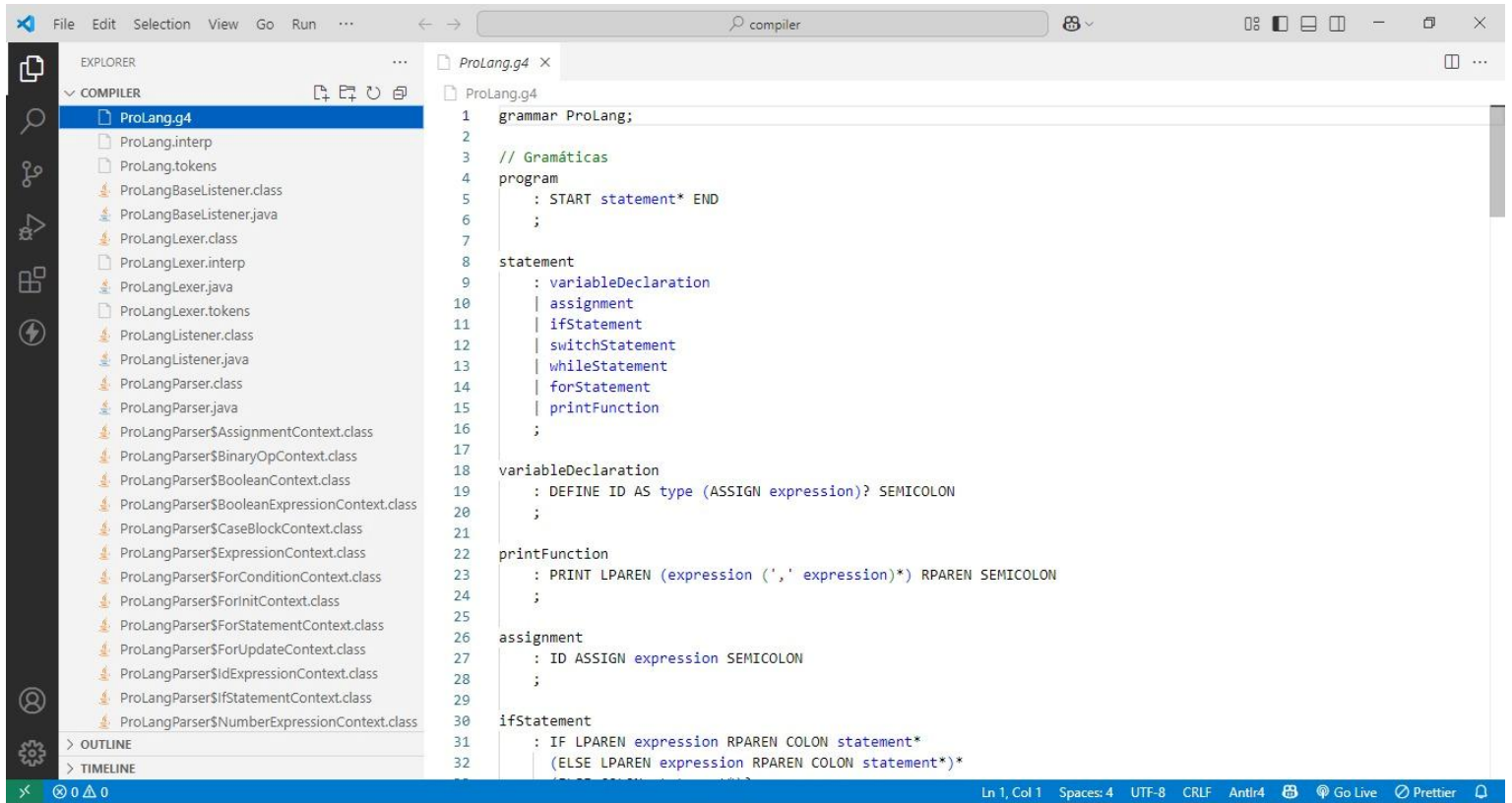
```

```

// Operadores de comparación
EQUAL_THAN : '==';

```

Prueba de elaboración.



The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer lists files under 'COMPILER', including 'ProLang.g4' and various listener and parser classes. The code editor displays the 'ProLang.g4' file, which contains a grammar definition for ProLang. The grammar includes rules for 'program', 'statement', 'variableDeclaration', 'printFunction', 'assignment', and 'ifStatement'. The status bar at the bottom indicates the current line and column (Ln 1, Col 1), the number of spaces (4), the encoding (UTF-8), the line ending (CRLF), and the active theme (Antlr4). The status bar also includes icons for 'Go Live' and 'Prettier'.

```
1 grammar ProLang;
2
3 // Gramáticas
4 program
5 : START statement* END
6 ;
7
8 statement
9 : variableDeclaration
10 | assignment
11 | ifStatement
12 | switchStatement
13 | whileStatement
14 | forStatement
15 | printFunction
16 ;
17
18 variableDeclaration
19 : DEFINE ID AS type (ASSIGN expression)? SEMICOLON
20 ;
21
22 printFunction
23 : PRINT LPAREN (expression (',' expression)*) RPAREN SEMICOLON
24 ;
25
26 assignment
27 : ID ASSIGN expression SEMICOLON
28 ;
29
30 ifStatement
31 : IF LPAREN expression RPAREN COLON statement*
32 (ELSE LPAREN expression RPAREN COLON statement*)*
```