

Universidad Mariano Gálvez De Guatemala
Sede De Boca Del Monte, Villa Canales
Ingeniería en Sistemas de Información
Ing. Ezequiel Urizar Araujo
Compiladores



Proyecto I
Analizador Léxico y Sintáctico

Nombre:	Carnet:	participación
Bryan Manuel Pineda Orozco	7690-16-8869	100%
Jonathan Joel Chán Cuellar	7690-22-1805	100%
Roberto Antonio Ramirez Gómez	7690-22-12700	100%
Jean Klaus Castañeda Santos	7690-22-892	100%
Edwin Rolando Ixcoy Tot	7690-16-1582	100%

Guatemala, 07 de marzo de 2025

Índice

Descripción del diseño del lenguaje.	3
Estructura General	3
Explicación de las reglas de la gramática.	4
Reglas semánticas	4
1. Declaraciones y Asignaciones	4
2. Sentencias Print	5
3. Estructuras Condicionales (If, Else If, Else)	5
4. Ciclos (While y For)	5
5. Expresiones	5
6. Bloques y Ámbito	6
7. Tipos Permitidos	6
8. Manejo de Errores	6
Reglas léxicas	6
Delimitadores y Símbolos	6
2. Palabras Clave	6
3. Operador de Asignación.....	7
4. Operadores Matemáticos	7
5. Operadores Relacionales y Lógicos	7
6. Operadores de Incremento y Decremento	8
7. Identificadores y Números	8
8. Espacios en Blanco	8
Ejemplos de Entrada y Salida.....	9
Pasos Realizados	11
Conclusión	15

Descripción del diseño del lenguaje.

Estructura General

La gramática define cómo se deben estructurar las instrucciones en el lenguaje por lo que hacemos la breve explicación y desglose de la siguiente gramática la cual el formato es de la siguiente manera lo cual acepta números enteros o decimales

- Gramática Principal:

VARIABLE LLAVE_APERTURA instruccion+ LLAVE_CIERRE EOF

Define que una gramática consiste en una o más instrucciones seguidas del final del archivo (EOF).

Ejemplo

Programa{

Todas las instrucciones

}

¿Por qué variable?: Porque puede iniciar con cualquier nombre variable que cumpla con la regla definida dentro de variable.

Puede ser una declaración, una sentencia print, una sentencia if, while o for.

- Declaraciones y Asignaciones

declaracion

Una declaración es una expresión de asignación seguida de un fin de línea.

asignacion_expr

Asigna un valor a una variable.

- Sentencias

sentencia_print

Imprime una expresión.

sentencia_if

Define una estructura if con posibles else if y else.

sentencia_while

Define un bucle while.

sentencia_for

- **Componentes del For**

Inicialización

for_inicializacion

Puede ser una asignación o estar vacío.

for_condicion

Puede ser una expresión o estar vacío.

Incremento:

Puede ser un incremento/decremento o una asignación.

Bloques y Expresiones

Puede ser un conjunto de instrucciones entre llaves o una sola instrucción.

Expresiones:

Define las operaciones aritméticas, relacionales y el uso de paréntesis, variables y números.

Tales como suma resta, multiplicación, división y potencias

Explicación de las reglas de la gramática.

Reglas semánticas

1. Declaraciones y Asignaciones

- Tipo de datos: Las variables deben ser declaradas antes de su uso. Además, las asignaciones deben respetar el tipo de dato; por ejemplo, no se puede asignar un valor numérico a una variable booleana.
- Inicialización: Una variable debe ser inicializada antes de ser usada en cualquier expresión.
- Asignación válida: La expresión a la derecha del operador = debe ser del mismo tipo que la variable a la izquierda.

Ejemplo válido:

x = 5; // Válido si x es numérica

2. Sentencias Print

- Compatibilidad de tipos: La expresión dentro de print debe ser de un tipo que pueda ser convertido a cadena (números, cadenas, booleanos).

Ejemplo válido:

```
print(x);    // Válido si x está inicializada
```

```
print(5 + 3); // Válido
```

3. Estructuras Condicionales (If, Else If, Else)

- Condición booleana: La expresión dentro de if, else if debe evaluar a un valor booleano.
- Compatibilidad de ramas: Todas las ramas deben manejar tipos compatibles. Si una rama retorna un tipo, todas deben hacerlo.

Ejemplo válido:

```
if (x > 5) {  
    print(1);  
} else {  
    print(0);  
}
```

4. Ciclos (While y For)

While

- Condición booleana: La expresión dentro de while debe evaluar a true o false.

For

- Inicialización, condición y actualización: La parte de inicialización debe declarar o asignar una variable. La condición debe ser booleana y la actualización debe modificar la variable de control.

Ejemplo válido:

```
for (int i = 0; i < 10; i++) {  
    print(i);  
}
```

5. Expresiones

- Operaciones aritméticas: Solo se permiten entre tipos numéricos.

- Operadores relacionales: Comparaciones como >, <, >=, <=, ==, != deben realizarse entre tipos compatibles.

6. Bloques y Ámbito

- Ámbito local: Variables declaradas dentro de un bloque {} solo son accesibles dentro de ese bloque.
- Ámbito global: Variables declaradas fuera de cualquier bloque son accesibles en todo el programa.

7. Tipos Permitidos

- int, float, bool, string

8. Manejo de Errores

- Errores sintácticos: Se deben detectar durante el análisis sintáctico.
- Errores semánticos: Se deben manejar durante el análisis semántico, como la incompatibilidad de tipos y el uso de variables no declaradas.

Conclusión

Las reglas semánticas descritas son fundamentales para asegurar la coherencia y la corrección del lenguaje de programación. Al garantizar que las variables se declaren y se utilicen adecuadamente, se evita la aparición de errores durante la ejecución del programa. Además, el manejo adecuado de los tipos de datos y el control de los ámbitos permiten una ejecución eficiente y segura. La implementación rigurosa de estas reglas facilita tanto el proceso de compilación como la detección temprana de errores, contribuyendo a la confiabilidad y mantenibilidad del código.

Reglas léxicas

Delimitadores y Símbolos

- **PARENTESIS_APERTURA** (()) **y** **PARENTESIS_CIERRE** ()): Se usan para delimitar agrupaciones por ejemplo: en expresiones o para encerrar condiciones en estructuras de control.
- **LLAVE_APERTURA** ({ }) **y** **LLAVE_CIERRE** (}): Indican el inicio y fin de un bloque de instrucciones similar a como se usan en lenguajes de programación
- **FIN_DE_LINEA** (;): Es el token que indica al compilador dónde termina una sentencia.

2. Palabras Clave

Estas reglas reconocen las palabras reservadas que definen la estructura del programa:

- **IF:** Reconoce la palabra reservada "if" para comenzar una condición.
- **ELSE:** Se utiliza para definir un bloque alternativo en estructuras condicionales.
- **WHILE:** Palabra reservada para iniciar un bucle que se repite mientras se cumpla una condición.
- **FOR:** Se emplea para definir un ciclo for, que incluye inicialización, condición e incremento.
- **PRINT:** Indica la instrucción para imprimir valores o resultados en la salida.

3. Operador de Asignación

- **ASIGNACION(=):**
Este token reconoce el símbolo igual, usado para asignar valores a variables.

4. Operadores Matemáticos

Estos tokens permiten realizar operaciones aritméticas en las expresiones:

- **MAS (+):** Sumar.
- **MENOS (-):** Restar.
- **MULTIPLICACION (*):** Multiplicación.
- **DIVISION (/):** División.
- **POW (^):** Representa la operación de potencia.

5. Operadores Relacionales y Lógicos

Sirven para comparar valores y definir condiciones:

- **IGUAL (==):** Compara dos valores para ver si son iguales.
- **DIFERENTE (!=):** Comprueba que dos valores no sean iguales.
- **MENOR (<):** Verifica si un valor es menor que otro.
- **MAYOR (>):** Verifica si un valor es mayor que otro.
- **MENOR_IGUAL_QUE (<=):** Compara si un valor es menor o igual que otro.
- **MAYOR_IGUAL_QUE (>=):** Compara si un valor es mayor o igual que otro.

6. Operadores de Incremento y Decremento

Estos tokens permiten modificar el valor de una variable de forma abreviada:

- **MASMAS (++)**: Incrementa el valor de una variable en una unidad.
- **MENOSMENOS (--)**: Decrementa el valor de una variable en una unidad.

7. Identificadores y Números

- **VARIABLE:**
Define el formato que deben seguir los nombres de las variables. La regla establece que debe comenzar con una letra (mayúscula o minúscula) o un guion bajo (`_`), seguido de cualquier combinación de letras, dígitos o guiones bajos. Esto asegura que los identificadores sean claros y consistentes.
- **NUMERO:**
Reconoce secuencias de dígitos que representan números enteros o decimales. Permite una parte entera seguida opcionalmente de un punto y una parte decimal. Esto es esencial para soportar tanto operaciones aritméticas simples como cálculos con números reales.

8. Espacios en Blanco

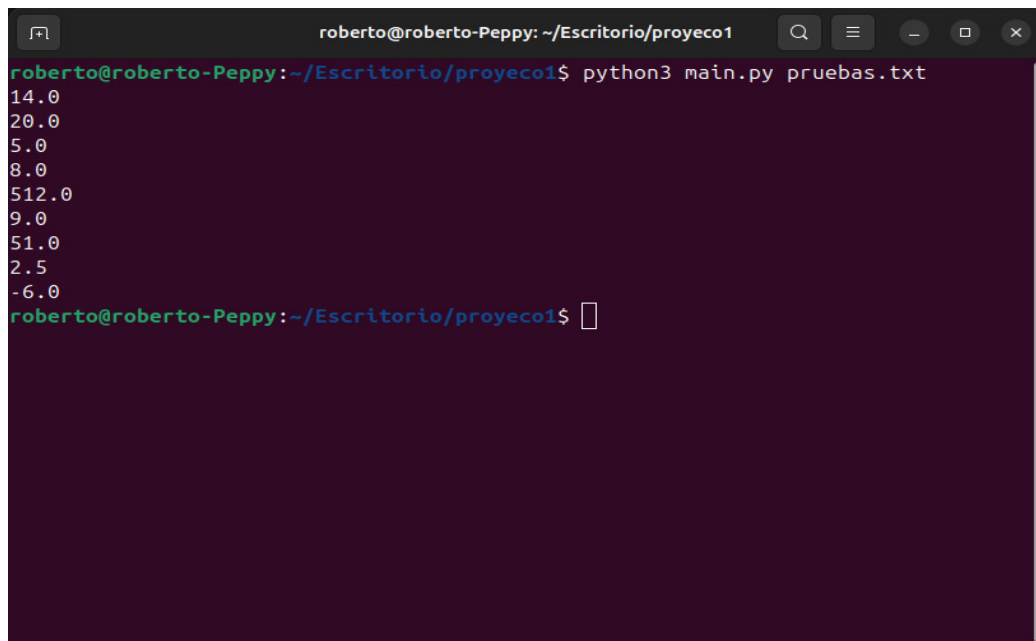
- **WS:**
La regla WS identifica los espacios, tabulaciones y saltos de línea. El operador `-> skip` indica que estos caracteres se omiten durante el análisis, ya que no son relevantes para la sintaxis del lenguaje. Esto ayuda a que el lexer se enfoque únicamente en los tokens significativos.

Ejemplos de Entrada y Salida

Ejemplo 1 – pruebas.txt

```
miprograma{  
    x = 3;  
    print(2 + x * 4);  
    print((2 + 3) * 4);  
    print(10 / 2);  
    print(2 ^ 3);  
    print(2 ^ 3 ^ 2);  
    print(5 + 3 * 2 - 4 / 2);  
    print((3 + 4) * 2 ^ 3 - 10 / 2);  
    print(5 / 2);  
    print(2 + 4 * (-2));  
}
```

Salida - Resultados

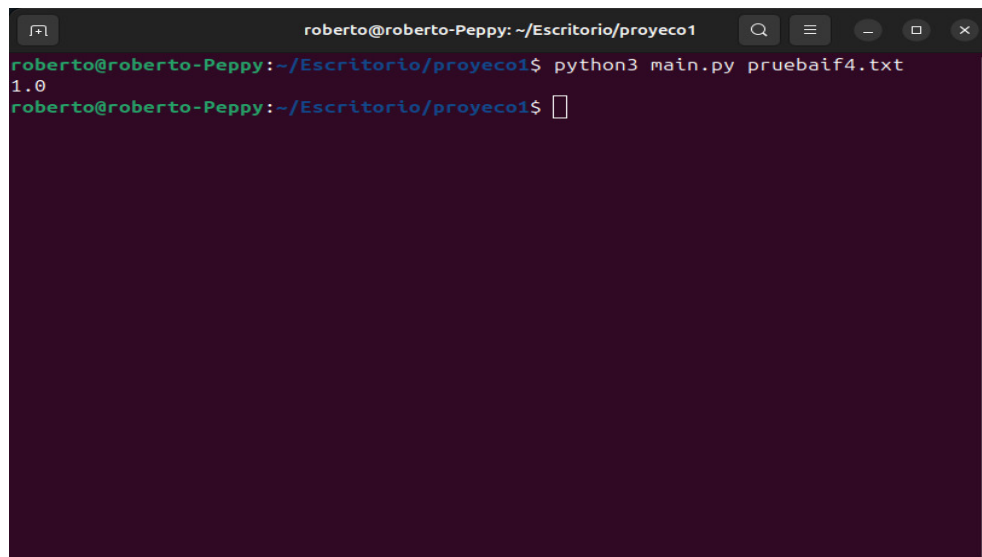
A screenshot of a terminal window with a dark background. The title bar at the top reads 'roberto@roberto-Peppy: ~/Escritorio/proyec01'. The terminal shows the command 'python3 main.py pruebas.txt' being executed. The output consists of ten lines of numerical results: 14.0, 20.0, 5.0, 8.0, 512.0, 9.0, 51.0, 2.5, -6.0, and a final prompt line. The prompt 'roberto@roberto-Peppy: ~/Escritorio/proyec01\$' is visible at the bottom of the output.

```
roberto@roberto-Peppy: ~/Escritorio/proyec01  
roberto@roberto-Peppy:~/Escritorio/proyec01$ python3 main.py pruebas.txt  
14.0  
20.0  
5.0  
8.0  
512.0  
9.0  
51.0  
2.5  
-6.0  
roberto@roberto-Peppy:~/Escritorio/proyec01$
```

Ejemplo 2 – pruebaif4.txt

```
programa{  
    x = 10;  
    if (x > 5) {  
        if (x < 15) {  
            print(1);  
        } else {  
            print(2);  
        }  
    } else {  
        print(3);  
    }  
}
```

Salida – Resultado

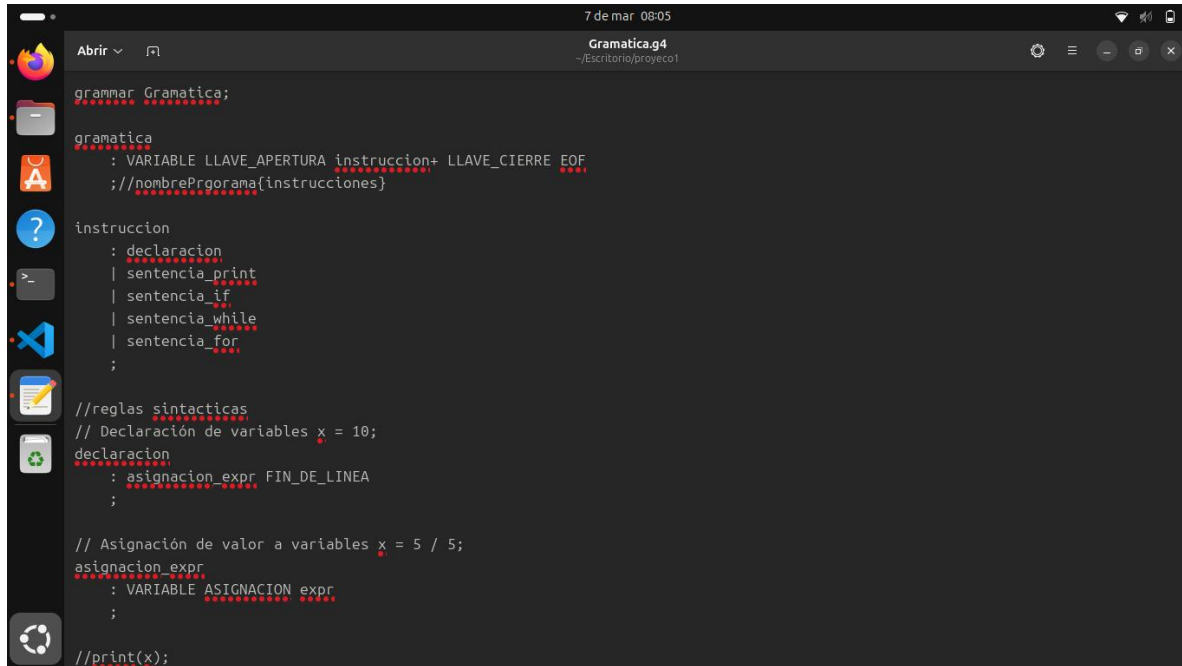


A terminal window with a dark background and light-colored text. The window title is "roberto@roberto-Peppy: ~/Escritorio/proyecto1". The prompt is "roberto@roberto-Peppy:~/Escritorio/proyecto1\$". The command "python3 main.py pruebaif4.txt" has been entered and executed. The output "1.0" is displayed on the line following the command. The prompt "roberto@roberto-Peppy:~/Escritorio/proyecto1\$" is shown again on the next line, followed by a cursor.

```
roberto@roberto-Peppy: ~/Escritorio/proyecto1  
roberto@roberto-Peppy:~/Escritorio/proyecto1$ python3 main.py pruebaif4.txt  
1.0  
roberto@roberto-Peppy:~/Escritorio/proyecto1$
```

Pasos Realizados

1. Instalación de ANTLR en sistema operativo Ubuntu.
2. Creación del archivo Gramatica.g4
3. Redactar las reglas sintácticas que definirán nuestro lenguaje en el archivo Gramatica.g4



```
grammar Gramatica;

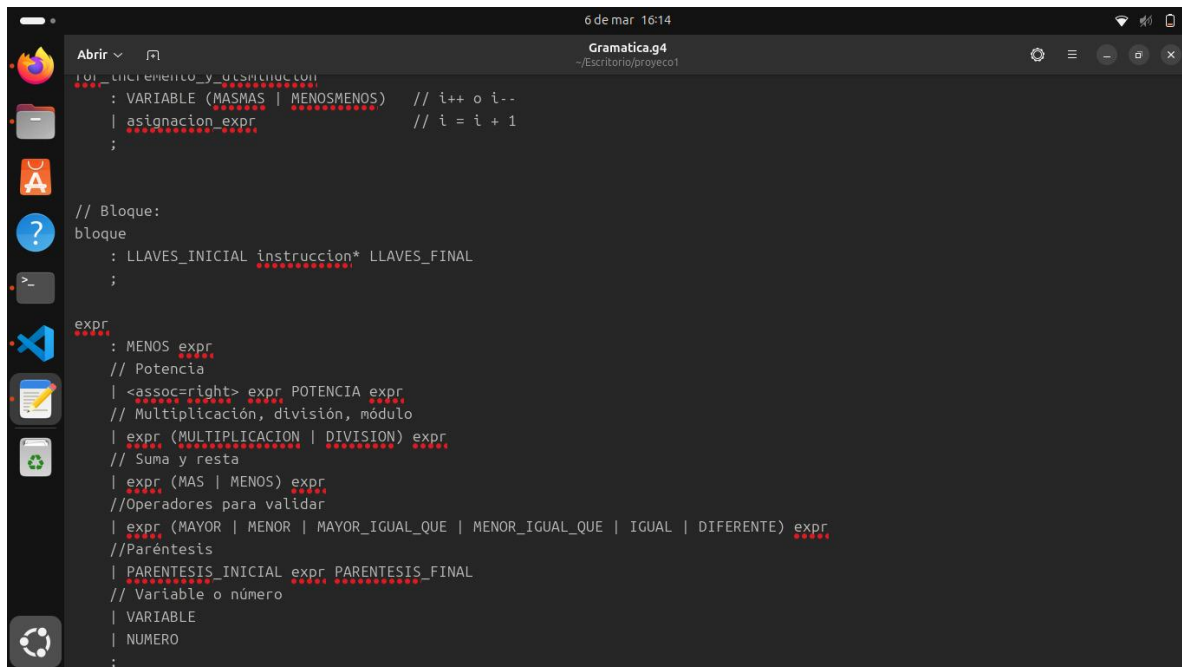
gramatica
: VARIABLE LLAVE_APERTURA instruccion+ LLAVE_CIERRE EOF
;

instruccion
: declaracion
| sentencia_print
| sentencia_if
| sentencia_while
| sentencia_for
;

//reglas sintacticas
// Declaración de variables x = 10;
declaracion
: asignacion_expr FIN_DE_LINEA
;

// Asignación de valor a variables x = 5 / 5;
asignacion_expr
: VARIABLE ASIGNACION expr
;

//print(x);
```

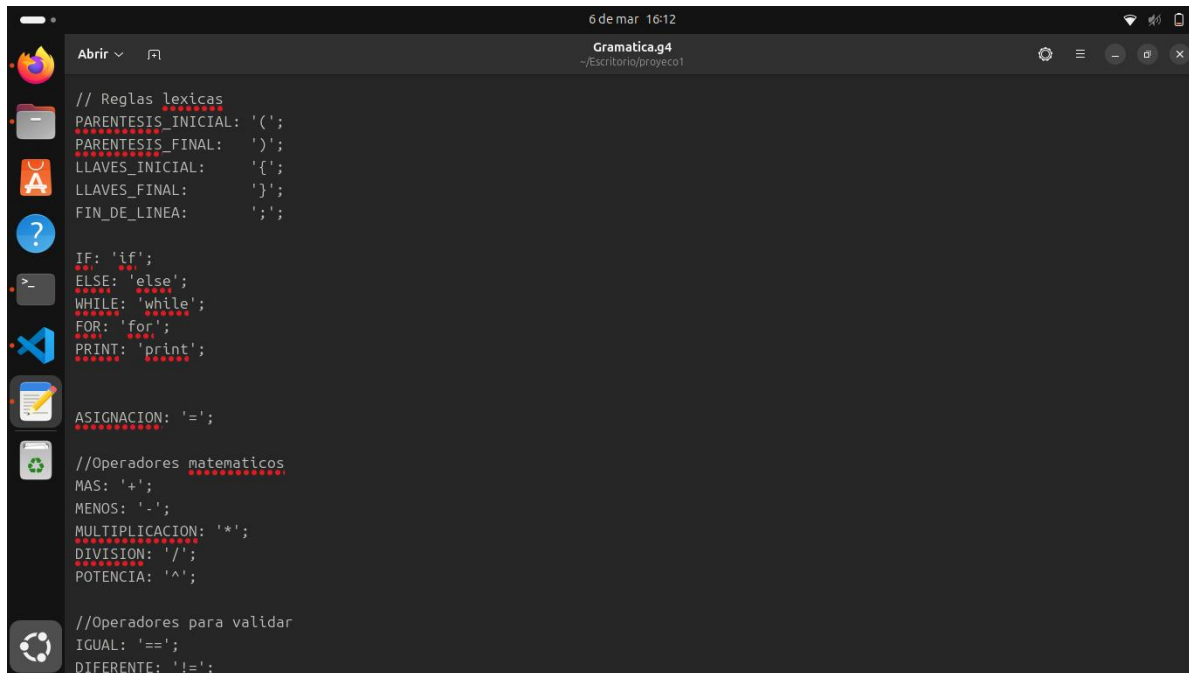


```
for _token_ en token y _q_ en instruccion
: VARIABLE (MASMAS | MENOSMENOS) // i++ o i--
| asignacion_expr // i = i + 1
;

// Bloque:
bloque
: LLAVES_INICIAL instruccion* LLAVES_FINAL
;

expr
: MENOS expr
// Potencia
| <assoc=right> expr POTENCIA expr
// Multiplicación, división, módulo
| expr (MULTIPLICACION | DIVISION) expr
// Suma y resta
| expr (MAS | MENOS) expr
//Operadores para validar
| expr (MAYOR | MENOR | MAYOR_IGUAL_QUE | MENOR_IGUAL_QUE | IGUAL | DIFERENTE) expr
//Paréntesis
| PARENTESIS_INICIAL expr PARENTESIS_FINAL
// Variable o número
| VARIABLE
| NUMERO
;
```

4. Redactar las reglas léxicas permitidas para nuestro lenguaje en el archivo Gramatica.g4



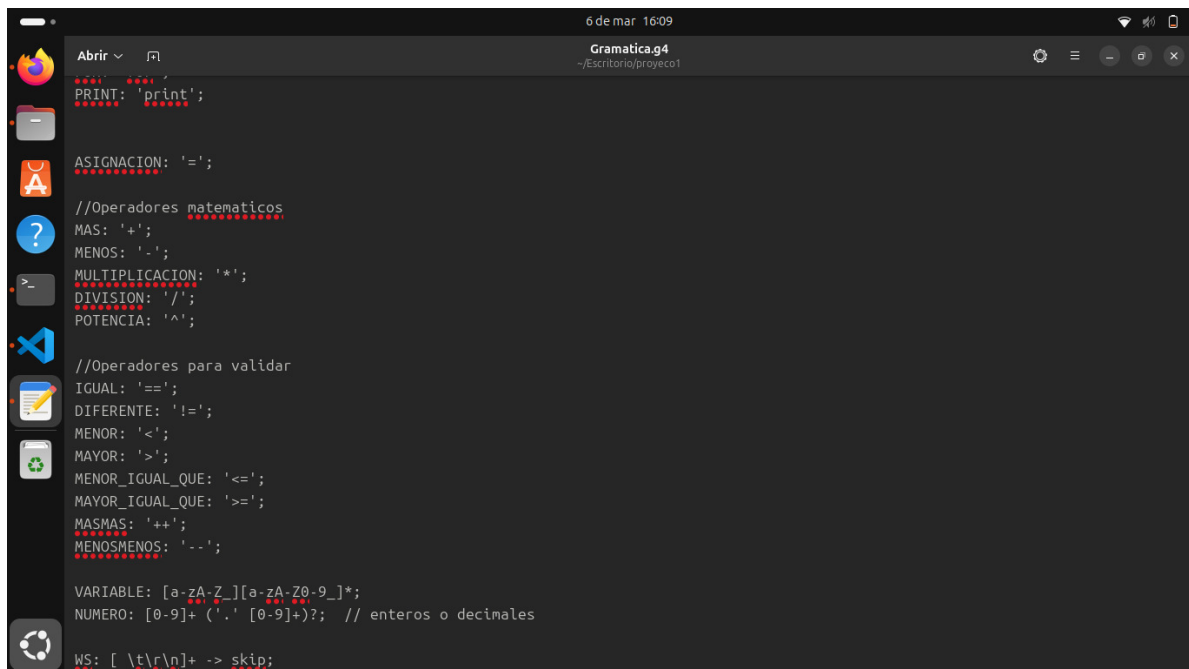
```
// Reglas léxicas
PARENTESIS_INICIAL: '(';
PARENTESIS_FINAL: ')';
LLAVES_INICIAL: '{';
LLAVES_FINAL: '}';
FIN_DE_LINEA: '\n';

IF: 'if';
ELSE: 'else';
WHILE: 'while';
FOR: 'for';
PRINT: 'print';

ASIGNACION: '=';

//Operadores matematicos
MAS: '+';
MENOS: '-';
MULTIPLICACION: '*';
DIVISION: '/';
POTENCIA: '^';

//Operadores para validar
IGUAL: '==';
DIFERENTE: '!=';
```



```
PRINT: 'print';

ASIGNACION: '=';

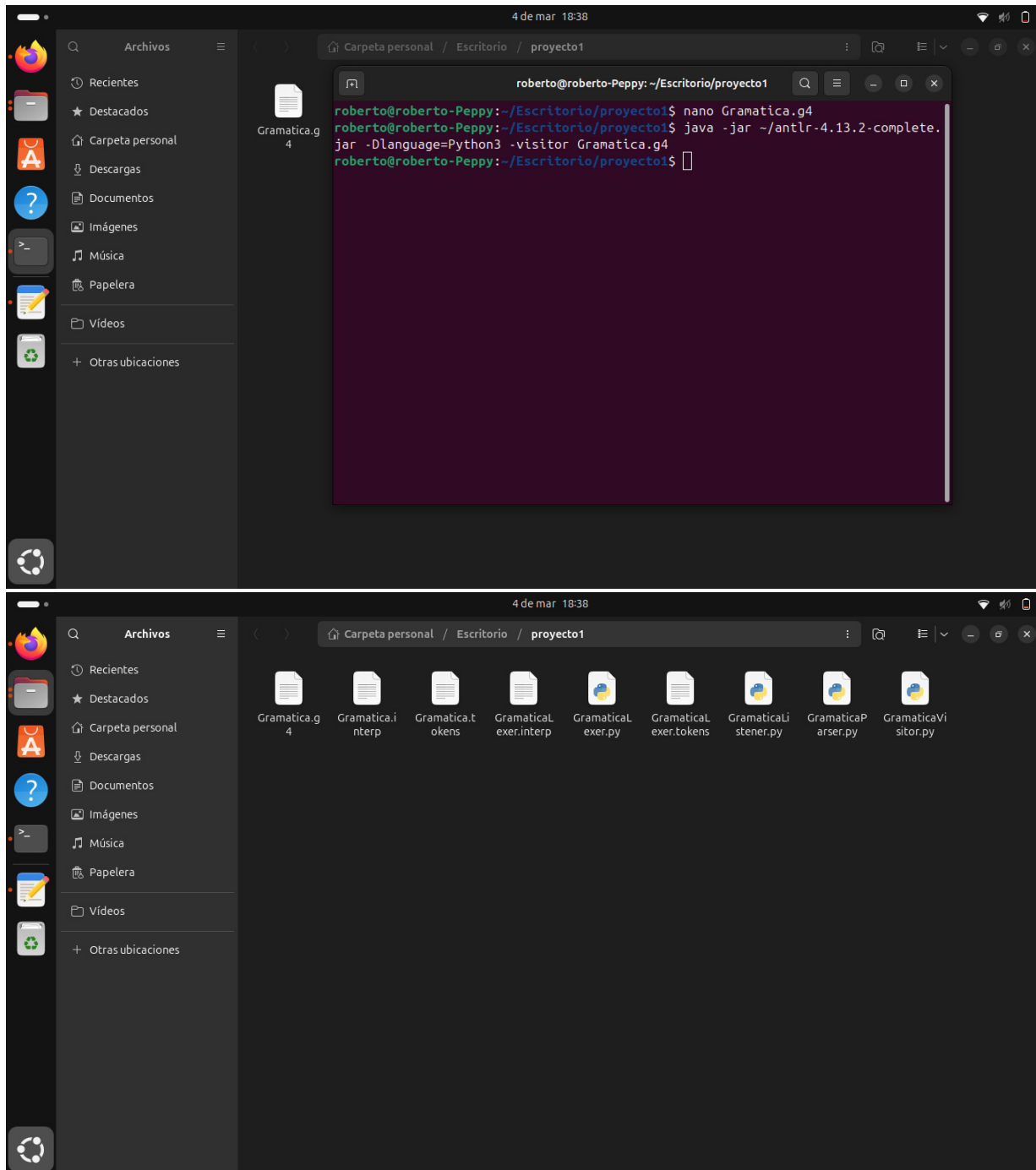
//Operadores matematicos
MAS: '+';
MENOS: '-';
MULTIPLICACION: '*';
DIVISION: '/';
POTENCIA: '^';

//Operadores para validar
IGUAL: '==';
DIFERENTE: '!=';
MENOR: '<';
MAYOR: '>';
MENOR_IGUAL_QUE: '<=';
MAYOR_IGUAL_QUE: '>=';
MASMAS: '++';
MENOSMENOS: '--';

VARIABLE: [a-zA-Z_][a-zA-Z0-9_]*;
NUMERO: [0-9]+ ('.' [0-9]+)?; // enteros o decimales

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

5. Guardar los cambios en el archivo y compilar el comando en la terminal de Linux para generar los archivos necesarios para la implementación en Python.



6. Implementación de los archivos en Python para la validación de lenguajes ingresados por medio de un archivo.txt

Link del repositorio GitHub con el código fuente

- Main.py
- Visitor.py
- Archivos generados por ANTLR
- Archivos de texto con pruebas realizadas

<https://github.com/rramirezg18/Proyecto1Compiladores>

Conclusión

El desarrollo del Proyecto de Compiladores ha permitido explorar en profundidad los aspectos fundamentales del análisis sintáctico y semántico en el proceso de compilación. A través de la implementación de un analizador sintáctico basado en gramáticas específicas y un análisis semántico riguroso, se logró asegurar que el código fuente cumpla tanto con las reglas estructurales como con las restricciones lógicas y de tipos definidas por el lenguaje.

La gramática definida permitió manejar correctamente las instrucciones, declaraciones, estructuras de control y expresiones, asegurando que el código ingresado siguiera una estructura clara y predecible. El uso de árboles sintácticos y la aplicación de reglas semánticas facilitaron la detección temprana de errores, como el uso indebido de tipos, variables no declaradas y condiciones incompatibles. Además, la inclusión de pruebas exhaustivas con casos tanto válidos como erróneos demostró la efectividad y robustez del analizador, validando su capacidad para manejar distintos escenarios de manera eficiente.

El análisis semántico reforzó la coherencia lógica del código, asegurando que las operaciones y asignaciones fueran compatibles en términos de tipos de datos. Esta etapa resultó crucial para evitar errores durante la ejecución y para mejorar la calidad general del software generado. Asimismo, las pruebas realizadas confirmaron que las reglas semánticas se aplicaban correctamente, garantizando la integridad de los datos y el flujo lógico del programa.

El proyecto no solo consolidó los conocimientos sobre teoría de compiladores y lenguajes formales, sino que también proporcionó una base práctica sólida para futuros desarrollos en el campo de los compiladores. La integración eficaz de los análisis sintáctico y semántico, junto con la validación a través de pruebas, demostró la importancia de un diseño bien estructurado y la relevancia de abordar cada etapa del proceso de compilación de manera meticulosa. Este enfoque garantiza que el código fuente no solo sea sintácticamente correcto sino también semánticamente coherente, contribuyendo a la creación de software confiable y eficiente.