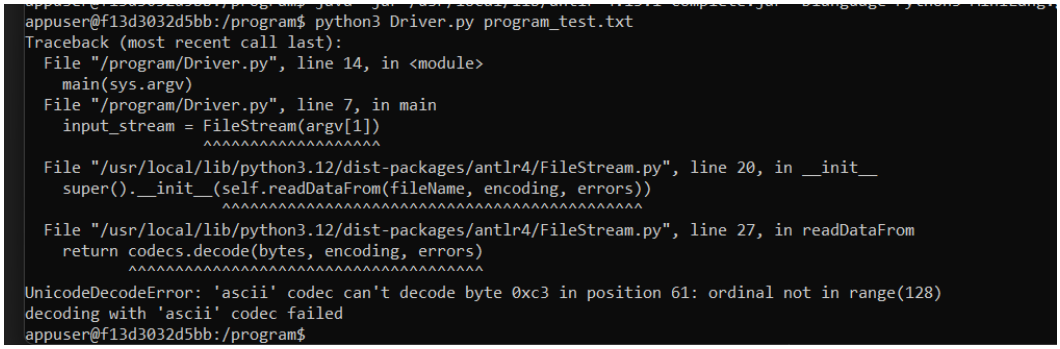
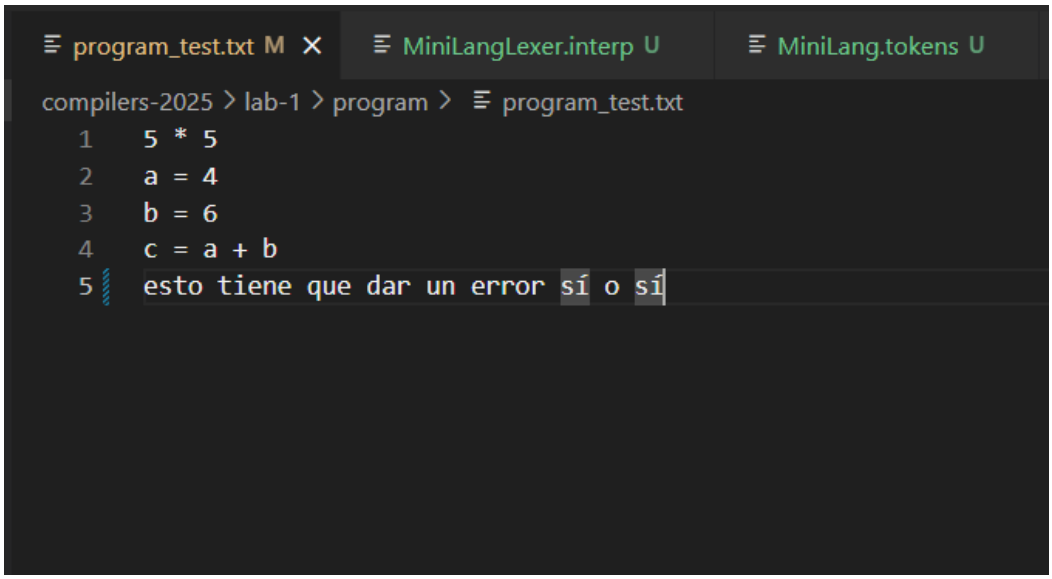
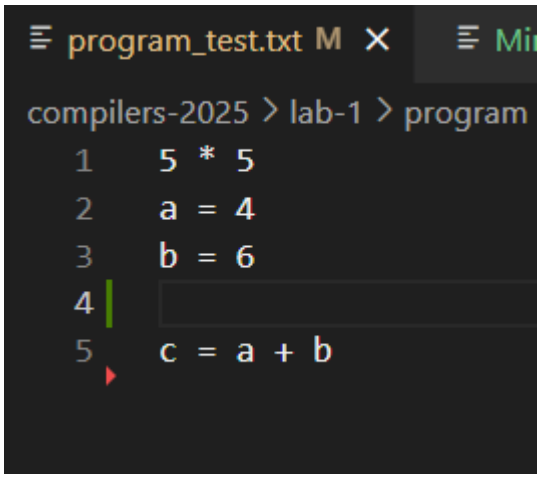
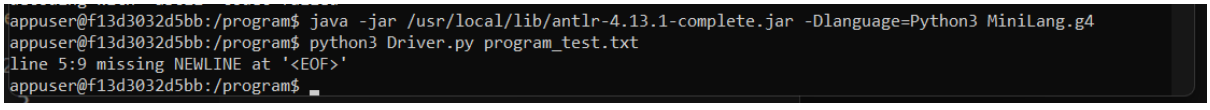
Irving Acosta 22781
Laboratorio#1
Gabriel Brolo-Sección 10









program_test.txt M MiniLangLexer.interp U

compilers-2025 > lab-1 > program > program_test.txt

```
1   5 * 5
2   a = 4
3   b = 6  c = a + b
```

```
appuser@f13d3032d5bb:/program$ python3 Driver.py program_test.txt
line 3:7 missing NEWLINE at 'c'
line 3:16 missing NEWLINE at '<EOF>'
appuser@f13d3032d5bb:/program$
```

compilers-2025 > lab-1 > program > program_test.txt

```
1   5 * 5
2   a = 4
3   b = 0
4   c = a / b
5   d = "a"
```

```
appuser@f13d3032d5bb:/program$ java -jar /usr/local/lib/antlr-4.13.1-comple
appuser@f13d3032d5bb:/program$ python3 Driver.py program_test.txt
line 5:4 token recognition error at: '"'
line 5:6 token recognition error at: '"'
line 5:7 missing NEWLINE at '<EOF>'
appuser@f13d3032d5bb:/program$
```

```
appuser@f13d3032d5bb:/program$ python3 Driver.py program_test.txt
line 5:6 no viable alternative at input 'print('
line 5:9 missing NEWLINE at '<EOF>'
appuser@f13d3032d5bb:/program$
```

program_test.txt M

compilers-2025 > lab-1 > progra

```
1  5 * 5
2  a = 4
3  b = 0
4  c = a / b
5  print (a)
```

```
appuser@f13d3032d5bb:/program$ python3 Driver.py program_test.txt
line 5:4 extraneous input '\r\n' expecting {'(', ID, INT}
line 6:2 mismatched input '=' expecting NEWLINE
appuser@f13d3032d5bb:/program$
```

lers-2025 > lab-1 > program >

```
5 * 5
a = 4
b = 0
a =
c = a + b - g
```

- Realice un análisis sobre la gramática de ANTLR y el archivo de Driver y comente acerca del funcionamiento de estos, es decir, explique sus partes lo más brevemente posible e indique cómo funcionan los distintos elementos de la gramática escrita en ANTLR, e.g. "Utilizar # en ANTLR sirve para...", "Un archivo .g4 tiene las siguientes secciones...", etc.

  La manera en qué funciona la gramática se explica a través del Driver.py
  Este archivo nos proporciona una vista al pipeline de los pasos que sigue para construir.

```
    input_stream = FileStream(argv[1])
    lexer = MiniLangLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = MiniLangParser(stream)
    tree = parser.prog()  # We are using 'prog' since this is the starting
rule based on our MiniLang grammar, yay!
```

1. La primera línea recibe el fichero de entrada para leer el fichero de entrada
2. La segunda se encarga de realizar la tokenización basandose en las reglas del archivo Minilang.g4
3. En la tercera línea se bufferizan los tokens
4. En la carta se prepara el parser usando el flujo de tokens
5. Por último, se llama a la regla inicial 'prog' de la gramática

Ahora bien, este es el flujo “general” que lleva, pero para entrar de lleno en las reglas que sigue podemos observar el archivo .g4, en donde encontramos las siguientes reglas:

`grammar MiniLang;`	Se declara el nombre de la grámatica
prog: stat+ ;	Es la regla inicial y sgnifica “una o más (+) sentencias (`stat`)”
Stat : expr NEWLINE # printExpr | ID '=' expr NEWLINE # assign | NEWLINE # blank ;	Los stats son las reglas gramaticales y en este caso pueden ser Una expresión seguida de salto de línea Una variable (un identificador), seguido de un signo “=” , una expresión y un salto de línea Un salto de línea
expr: expr ('*'|'/') expr # MulDiv | expr ('+'|'-') expr # AddSub | INT # int | ID # id	Luego pasamos a la parte de expr, donde es recursiva para manejar operaciones con precedencia

| '(' expr ')' # parens ;	**MulDiv**: `expr * expr` o `expr / expr` **AddSub**: `expr + expr` o `expr – expr` **int**: un literal entero (INT) **id**: una variable (ID) **parens**: sub-expresión entre paréntesis

MUL : '*' ; // define token for multiplication
DIV : '/' ; // define token for division
ADD : '+' ; // define token for addition
SUB : '-' ; // define token for subtraction
ID : [a-zA-Z]+ ; // match identifiers
INT : [0-9]+ ; // match integers
NEWLINE:'\r'? '\n' ; // return newlines to parser (is end-statement signal)
WS : [\t]+ -> skip ; // toss out whitespace
	Aquí se definen como se agrupan caracteres en tokens.
	Se escriben en mayúscula y no usan `:`; usan `->` para acciones.
	
	No voy a explicar cada token, porque ahí está en sus comentarios.

Entonces, la manera en que funciona es:

primero el lexer divide el texto en tokens

el paraser aplica las reglas declaradas en prog, stat, expr y construye un Árbol de sintaxis según la gramática

basandose en las etiquetas y clases generadas se puede diferenciar en el códgo los nodos y tomar decisiones basadas en ello.

- **Video de YouTube no listado** (pero público) con sus pruebas, donde compila bien y donde no compila bien y con sus comentarios al punto anterior.

  https://www.youtube.com/watch?v=k595FS3Pd4M

- Repo de Github con todo su código.