



Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Compiladores e intérpretes

Proyecto 2

Profesor

Emmanuel Ramírez Segura

Subgrupo 5

Estudiantes

Antonio Fernández García - 2022075006

Marco Rodriguez Vargas - 2022149445

Josué Mena González - 2022138381

II Semestre 2024

Índice

Gramática.....	3
Herramienta usada.....	5
Archivos generados automáticamente por ANTLR.....	9
Parte léxica.....	10
Parte Sintáctica.....	12
Parte Semántica.....	14
Logros/Fallos.....	18
Referencias.....	20

Gramática

Lenguaje L2 – Especificación Formal

programa \rightarrow declaraciones

declaraciones \rightarrow declaracion ; declaraciones | ϵ

declaracion \rightarrow expresion | control_flujo | impresion

control_flujo \rightarrow if_stmt | while_stmt | for_stmt

if_stmt \rightarrow if (expresion) { declaraciones } | if (expresion) { declaraciones } else { declaraciones }

while_stmt \rightarrow while (expresion) { declaraciones }

for_stmt \rightarrow for (expresion; expresion; expresion) { declaraciones }

impresion \rightarrow print (identificador)

expresion \rightarrow identificador = expresion | identificador | numero | (expresion)
| expresion operador_aditivo expresion | expresion operador_multiplicativo expresion
| expresion operador_relacional expresion | - expresion

operador_aditivo \rightarrow + | -

operador_multiplicativo \rightarrow * | /

operador_relacional \rightarrow < | > | == | <> | <= | >=

numero \rightarrow digito numero | digito

digito \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

identificador \rightarrow letra identificador | letra

letra \rightarrow a | b | c | ... | z

Lenguaje L2 – Especificación Informal

o Identificador

Los identificadores son secuencias de una o más letras minúsculas, hasta un máximo de 12 letras.

o ws (White Spaces)

Los espacios en blanco (tabs, espacios, enters) serán tomados como elementos separadores de los terminales.

Será por lo tanto válido cualquiera de las siguientes notaciones:

✓ Válido: `a=b+c;`

✓ Válido: `a = b + c ;`

✓ Válido: `a = b + c;`

***NOTA IMPORTANTE**

Vale la pena destacar que por la naturaleza de esta gramática se puede notar que la regla declaraciones produce una declaración seguida de “;” esto nos indica que siempre al terminar la declaración de un control de flujo este debe terminar con un “;”, para que se válido, por ejemplo: `if(expresion){`
 declaraciones
`};`

Esa es la forma correcta de declararlo según la gramática dada

*

Herramienta usada

Para el desarrollo de este proyecto se decidió utilizar ANTLR4 como herramienta generadora del analizador léxico y el analizador sintáctico. En este caso la herramienta podría generar las clases visitantes para el desarrollo del análisis semántico, pero en vez de optar por esto se tomó la decisión de utilizar el BaseListener que genera por defecto la herramienta. El listener en pocas palabras hace un recorrido más sencillo del AST que el Visitante y en este caso da dos métodos por cada regla de producción: uno de entrada y otro de salida. El listener, no permite manejar la forma en que se va recorriendo el árbol ni tampoco permite que sus métodos puedan retornar algo, pero es especialmente útil por su sencillez y por cómo permite acceder a los nodos de una forma relativamente simple.

Entendiendo mejor la herramienta a partir de búsquedas e investigación sobre cómo se puede recorrer el árbol que genera fue bastante sencillo hacer la implementación del análisis semántico. El proceso para poder usar la herramienta es el siguiente:

Crear el archivo .g4

Se crea un archivo .g4 que contiene la gramática especificada en el enunciado de este mismo proyecto que se ve así:

```
grammar l2;
programa: declaraciones;
declaraciones: declaracion';' declaraciones
              | ;
declaracion: expresion
            | control_flujo
            | impresion
            ;
control_flujo: if_stmt
              | while_stmt
              | for_stmt
              ;
if_stmt: 'if' '(' expresion ')' '{' declaraciones '}'
        | 'if' '(' expresion ')' '{' declaraciones '}' 'else' '{'
        declaraciones '}';
while_stmt: 'while' '(' expresion ')' '{' declaraciones '}'
```

```

;

for_stmt: 'for' '(' expresion ';' expresion ';' expresion ')' '{'
declaraciones '}';
impresion: 'print' '(' identificador ')';
expresion: identificador '=' expresion
          | identificador
          | numero
          | '(' expresion ')'
          | expresion operador_aditivo expresion
          | expresion operador_multiplicativo expresion
          | expresion operador_relacional expresion
          | '-' expresion
;

operador_aditivo: '+'
                | '-'
                ;

operador_multiplicativo: '*'
                       | '/'
                       ;

operador_relacional: '<'
                   | '>'
                   | '<='
                   | '>='
                   | '=='
                   | '<>'
                   ;

numero: digito numero
      | digito
      ;
digito: '0'
      | '1'
      | '2'
      | '3'
      | '4'
      | '5'
      | '6'
      | '7'
      | '8'

```

```
| '9'  
;
```

```
identificador: letra identificador  
              | letra  
              ;
```

```
letra: 'a'  
      | 'b'  
      | 'c'  
      | 'd'  
      | 'e'  
      | 'f'  
      | 'g'  
      | 'h'  
      | 'i'  
      | 'j'  
      | 'k'  
      | 'l'  
      | 'm'  
      | 'n'  
      | 'o'  
      | 'p'  
      | 'q'  
      | 'r'  
      | 's'  
      | 't'  
      | 'u'  
      | 'v'  
      | 'w'  
      | 'x'  
      | 'y'  
      | 'z'  
      | 'A'  
      | 'B'  
      | 'C'  
      | 'D'  
      | 'E'  
      | 'F'  
      | 'G'  
      | 'H'  
      | 'I'  
      | 'J'
```

```
| 'K'  
| 'L'  
| 'M'  
| 'N'  
| 'O'  
| 'P'  
| 'Q'  
| 'R'  
| 'S'  
| 'T'  
| 'U'  
| 'V'  
| 'W'  
| 'X'  
| 'Y'  
| 'Z'  
;
```

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

Ejecutar la herramienta

```
antlr4 12.g4
```

Este comando genera las clases necesarias para el compilador.








































Compilar los archivos

```
//Para Linux  
javac -cp ./compilacion/antlr-4.13.2-complete.jar *.java  
//Para Windows  
javac -cp .;compilacion/antlr-4.13.2-complete.jar *.java
```

Ejecutar el programa

```
Para Linux  
java -cp ./compilacion/antlr-4.13.2-complete.jar Main  
Para windows  
java -cp .;compilacion/antlr-4.13.2-complete.jar Main
```


Archivos generados automáticamente por ANTLR

 AnalizadorSemantico.class	 AnalizadorSemantico.java	 compilacion	 ejemploComplejo.txt	 errorFaseLexica.txt	 errorFaseSemantica.txt	 errorFaseSintactica.txt	 input.txt
 l2.g4	 l2.interp	 l2.tokens	 l2BaseListener.class	 l2BaseListener.java	 l2Lexer.class	 l2Lexer.interp	 l2Lexer.java
 l2Lexer.tokens	 l2Listener.class	 l2Listener.java	 l2Parser.class	 l2Parser.java	 l2Parser\$ControlFlujoContext.class	 l2Parser\$DeclaracionContext.class	 l2Parser\$DeclaracionesContext.class
 l2Parser\$DigitoContext.class	 l2Parser\$ExpresionContext.class	 l2Parser\$ForstmtContext.class	 l2Parser\$IdentificadorContext.class	 l2Parser\$IfstmtContext.class	 l2Parser\$ImpresionContext.class	 l2Parser\$LetraContext.class	 l2Parser\$NumeroContext.class
 l2Parser\$OperadorAditivoContext.class	 l2Parser\$OperadorMultiplicativoContext.class	 l2Parser\$OperadorRelacionalContext.class	 l2Parser\$ProgramaContext.class	 l2Parser\$WhilestmtContext.class	 Main.class	 Main.java	

Parte léxica

ANTLR4 hace esta parte de forma automática. Este como primer paso es muy importante para generar el resto del programa. El algoritmo utilizado para esta fase es LL(*), siendo uno capaz de elegir cuántos tokens de lookahead se pueden tomar. Por default esta herramienta trabaja con $k = 1$ para esto, y así fue como se trabajó para este proyecto.

Por motivos de sencillez y no complicar este documento no se va a mostrar el archivo generado, ya que este tiene más de 300 líneas de código y trabaja de una forma no del todo trivial. Finalmente, se llegó a la generación de esta parte y todo funciona de forma correcta y sin problemas, solo que existe un detalle con el analizador léxico y es el siguiente: Al existir palabras reservadas e identificadores este a veces confunde su autómata de reconocimiento de tokens entonces digamos que se tiene una variable así:

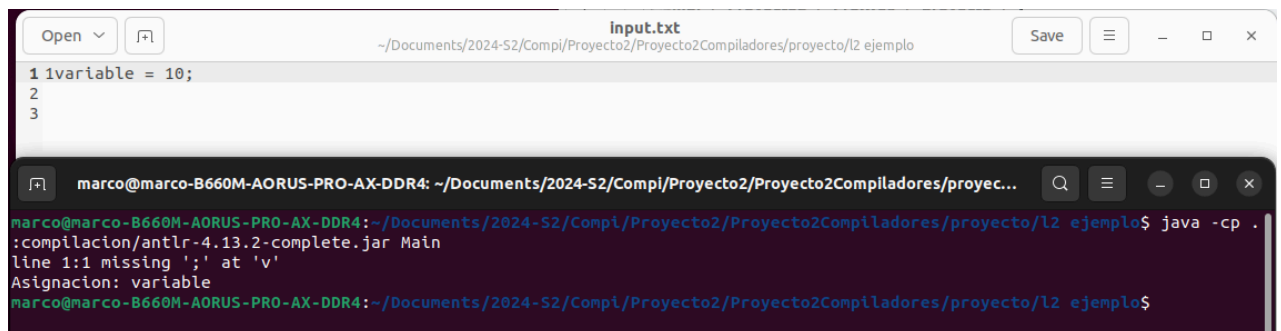
```
variableifrrara = 9;
```

Cuando se topa ese “if” el lexer lo toma como la palabra reservada, a pesar de venir como parte de un identificador. Este problema nunca se pudo solucionar ya que es parte de la generación automática.

Prueba con error léxico

1variable = 10;

Explicación: El error léxico ocurre debido a que el lexema no es válido. El identificador 1variable empieza con un número. lo que es invalido según la gramática y como se definió el identificador. La regla de la gramática estipula que debe iniciar con una letra A-Z o a-z. no es posible que sea un número.



The screenshot shows a code editor window titled 'input.txt' with the following content:

```
1 1variable = 10;
2
3
```

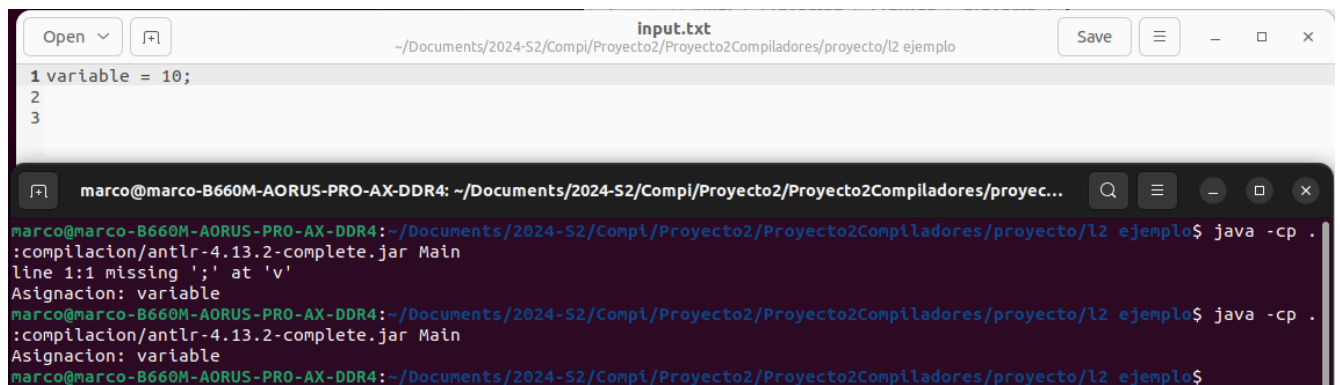
Below the editor is a terminal window. The command executed is `java -cp .:compilacion/antlr-4.13.2-complete.jar Main`. The output shows a lexical error:

```
line 1:1 missing ';' at 'v'
Asignacion: variable
```

Prueba fase léxica

variable = 10;

Explicación: Se soluciona el error léxico cuando el lexema empieza con una letra A-Z o a-z. El lexema variable es un identificador válido ya que cumple con las reglas de la gramática en la que se estipula debe iniciar con una letra A-Z o a-z. Luego la expresión a la derecha es válida porque 10 es un número y termina correctamente con punto y coma.



The screenshot shows the same code editor window with the corrected code:

```
1 variable = 10;
2
3
```

The terminal window shows the same command as before, but the output is now successful:

```
Asignacion: variable
```

Parte Sintáctica

Esta parte también es generada de forma automática y se hace todo de forma correcta. En realidad no da ningún error extraño y en caso de escribir algún tipo de código sospechoso realiza muy bien la identificación del error basado en la cadena de tokens que recibe, sea este un “;” faltante o un “}” faltante o un problema con la forma que viene escrito en general el código.

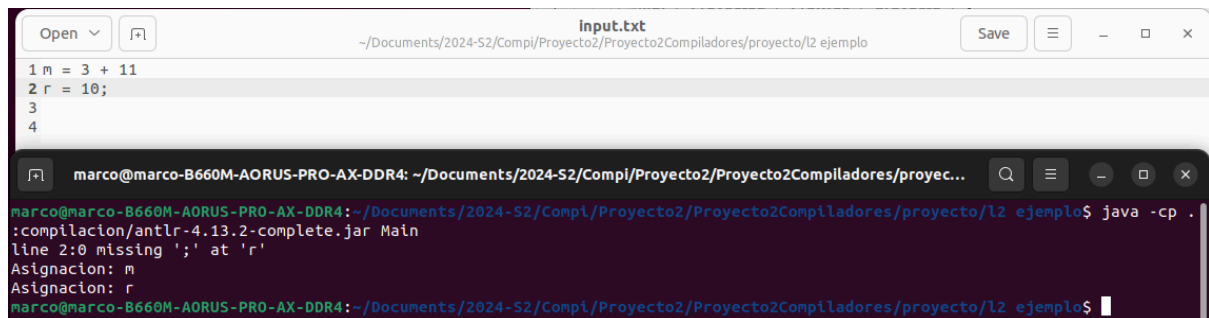
Es importante destacar que el programa no se detiene e incluso hace la parte semántica a pesar de encontrar un error, pero esto ya es por cómo la herramienta está construida y eso no se puede cambiar del todo. Como en la parte anterior, tampoco se mostrará el código completo de la fase sintáctica, ya que son muchas líneas de código y este no es fácil de comprender. Se toma en cuenta que esta parte está correctamente implementada y se sigue adelante.

Prueba con error sintáctico

m = 3 + 11

r = 10;

Explicación: El error sintáctico ocurre porque en la primera línea de la entrada m = 3 + 11 no termina con punto y coma. Según la gramática cada declaración debe terminar con punto y coma para ser válida. Por lo que la ausencia del punto y coma en m = 3 + 11 provoca un error sintáctico



The screenshot shows a code editor window titled 'input.txt' with the following content:

```
1 m = 3 + 11
2 r = 10;
3
4
```

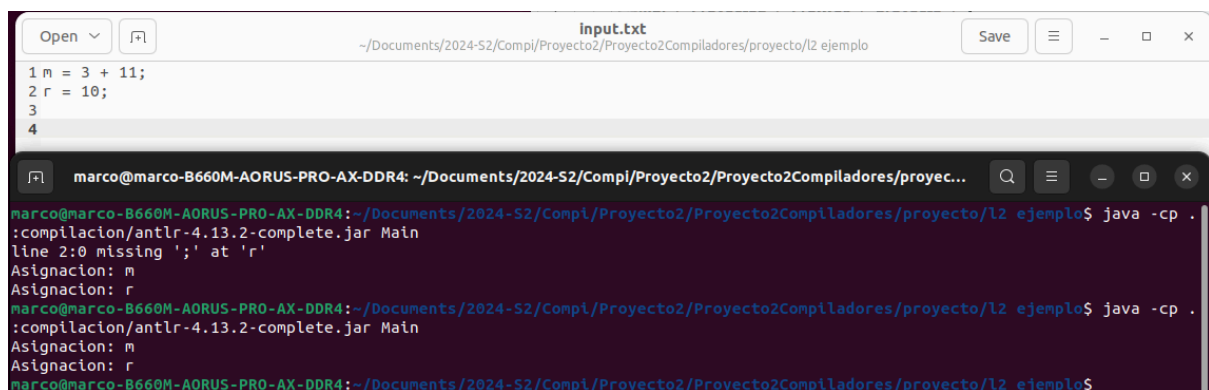
Below the editor is a terminal window showing the command `java -cp .:compilacion/antlr-4.13.2-complete.jar Main` being executed. The output shows the error message: `line 2:0 missing ';' at 'r'`. The terminal also shows the assignment of `m` and `r`.

Prueba fase sintáctica

m = 3 + 11;

r = 10;

Explicación: Se soluciona el error sintáctico cuando cada línea termina correctamente con punto y coma. En esta entrada tanto m = 3 + 11; como r = 10; terminan correctamente en punto y coma.



The screenshot shows the same code editor window as before, but with the following content:

```
1 m = 3 + 11;
2 r = 10;
3
4
```

The terminal window shows the command `java -cp .:compilacion/antlr-4.13.2-complete.jar Main` being executed. The output shows the error message: `line 2:0 missing ';' at 'r'`. The terminal also shows the assignment of `m` and `r`.

Parte Semántica

La parte semántica consiste en verificar dos aspectos muy importantes: que las variables están declaradas al ser usadas y evitar las divisiones por cero. Esto se hace por medio del AST que genera el mismo ANTLR4 y se hace un recorrido pero para esto se usa el Listener. Como fue mencionado anteriormente, en el listener cada regla tiene dos métodos; un enter y un exit. En este caso se trabaja así bajo estos pasos:

1. Se usa el enterExpresión ya que es aquella regla que puede tener como hijos en el árbol las diferentes formas donde se puede encontrar estos errores mencionados.
2. Se verifica que una expresión no es nula y esta tiene más de un hijo.
3. Si su segundo hijo es un símbolo "=" se procede a revisar si el identificador que está contenido en el nodo que le precede tiene un tamaño mayor a 12 en cuyo caso se lanza un error y se sale. Caso contrario se asigna la variable y se guarda en una tabla.
4. Luego está el caso donde un id está al otro lado del igual (derecha). Se recorren los hijos del nodo expresión a partir del "=" con un for loop. Cuando se encuentra un nodo identificador que se hace match por medio de una expresión regular se procede a revisar la tabla y ver si está declarado. En caso de no estarlo se retorna el error.
5. Como último caso está la división por 0. Es muy parecido al del punto 3. Si una expresión tiene como segundo hijo un "/" se procede en este caso a revisar el nodo que le sigue y verificar si tiene un 0 para lanzar el error.

Cabe destacar que la verificación del tamaño del identificador debió ser parte del analizador léxico, pero no se encontró forma de hacer que la herramienta lo limitara. Se investigó y en un foro de internet se dejó claro que la mejor forma de hacer esta verificación es desde el listener y fue esto lo que se terminó haciendo.

Códigos utilizados fase semántica

```
import java.util.HashSet;
import java.util.Set;

public class AnalizadorSemantico extends l2BaseListener{

    private Set<String> variables = new HashSet<>();

    public Set<String> getVariables() {
        return variables;
    }

    @Override
    public void enterExpresion(l2Parser.ExpresionContext ctx) {
        //si la expresion tiene un identificador se declara en la lista de variables
        //se verifica primera que hay una expresion y esta tiene hijos asociados en el arbol
        if(ctx.expresion() != null && ctx.getChildCount() > 1){
            //cuando el identificador esta al lado izquierdo del igual
            if(ctx.getChild(1).getText().equals("=")){
                //revision del largo del nombre del identificador
                if (ctx.identificador().getText().length() > 12) {
                    System.out.println("Nombre de identificador muy largo: " +
ctx.identificador().getText());
                    System.exit(1);
                }
                System.out.println("Asignacion: " + ctx.identificador().getText());
                String nombre = ctx.identificador().getText(); // se saca el nombre del identificador
                variables.add(nombre);
            }
            //cuando el identificador esta al lado derecho del igual
            // se recorren los hijos de la expresion y se verifica si son identificadores, si lo son
            // se verifica si estan declarados
            for(int i = 1; i < ctx.getChildCount(); i++){
                if(ctx.getChild(i) instanceof l2Parser.ExpresionContext){

                    String REGEX = "[a-zA-Z]+"; // una expresion que es un identificador
                    String nodo = ctx.getChild(i).getText();
                    boolean esVariable = nodo.matches(REGEX);
                    if(esVariable){
                        String nombre = ctx.getChild(i).getText();
                        if(!variables.contains(nombre)){
                            System.out.println("Error semantico: variable no declarada " + nombre);
                            System.exit(1);
                        }
                    }
                }
            }
        }
    }

    //chequeo division por cero
```

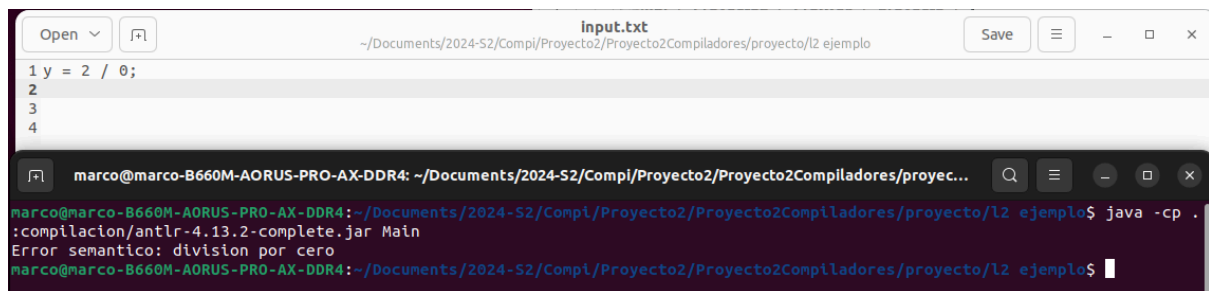
```
//si el operador es una division se verifica si el nodo derecho es 0
if(ctx.getChild(1).getText().equals("/")){

String nodo = ctx.getChild(2).getText(); //nodo derecho de la division
if(nodo.equals("0")){
    System.out.println("Error semantico: division por cero");
    System.exit(1);
}
}
}
}
```


Prueba con error semántico

$y = 2 / 0;$

Explicación: El error semántico ocurre debido a la división por cero, la cual es una operación indefinida que no está permitida en L2. En la línea $y = 2 / 0;$ el error se genera debido al intento de división de 2 entre 0, esto no tiene valor válido por lo que genera un error semántico.



The screenshot shows a code editor window titled 'input.txt' with the following content:

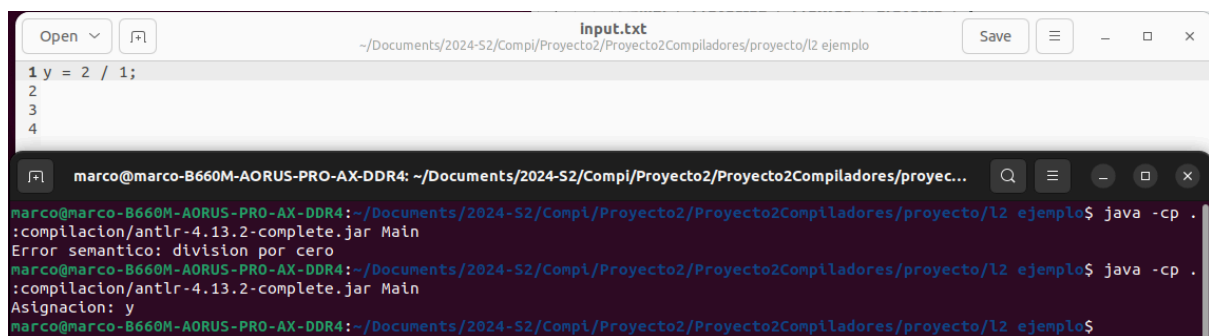
```
1 y = 2 / 0;  
2  
3  
4
```

Below the editor is a terminal window. The prompt is 'marco@marco-B660M-AORUS-PRO-AX-DDR4: ~/Documents/2024-S2/Compi/Proyecto2/Proyecto2Compiladores/proyec...'. The command executed is 'java -cp . :compilacion/antlr-4.13.2-complete.jar Main'. The output is 'Error semantico: division por cero'.

Prueba fase semántica

$y = 2 / 1;$

Explicación: Se soluciona el error semántico cuando no existe una división por cero. La operación de división de 2 entre 1 es válida porque no se está dividiendo entre cero. El valor de tanto el dividendo como el divisor son número enteros diferentes a cero. El identificador y es asignado correctamente con el valor de 2 entre 1 ya que la operación es válida.



The screenshot shows a code editor window titled 'input.txt' with the following content:

```
1 y = 2 / 1;  
2  
3  
4
```

Below the editor is a terminal window. The prompt is 'marco@marco-B660M-AORUS-PRO-AX-DDR4: ~/Documents/2024-S2/Compi/Proyecto2/Proyecto2Compiladores/proyec...'. The command executed is 'java -cp . :compilacion/antlr-4.13.2-complete.jar Main'. The output is 'Asignacion: y'.

Logros/Fallos

Al ser un proyecto relativamente grande que implicaba el uso de una herramienta nueva, se tuvo que trabajar por partes, para de esta forma asegurar que cada fase del proyecto funcionara correctamente y se comprendiera a fondo el funcionamiento y los métodos de la herramienta, pues posteriormente se debía trabajar una fase de manera manual y la integración era importante.

Se completó con éxito la fase léxica, para esto se escribió la gramática del proyecto de manera que ANTLR la entendiera, una vez hecho esto la herramienta generó las clases y archivos necesarios para realizar el análisis léxico. Cabe resaltar que la verificación del tamaño de las variables (máximo 12 letras) no se logró hacer en esta fase sino en la semántica, principalmente porque no existe una forma de señalar en la gramática que un identificador tenga un largo específico. También cabe destacar el punto que los identificadores no pueden contener las palabras reservadas del lenguaje dentro de ellos como el caso: **xxifxx = 9**; Esto forma parte de un fallo que se encontró con la herramienta.

Después, se logró completar exitosamente la fase sintáctica, en esta ANTLR generó clases para verificar que lo ingresado como entrada cumpliera con las reglas de producción especificadas en la gramática, además en este paso se logró generar el árbol AST, pieza clave para el análisis semántico.

Una vez llegado a la fase semántica la herramienta no era capaz de ayudar, por esta razón fue necesario realizar el análisis de forma manual, en este se logró comprender cómo ANTLR construía el AST, algo necesario pues se tenía que recorrer y hacer verificaciones con sus nodos. Las comprobaciones logradas fueron las necesarias para que el proyecto cumpliera con lo especificado, es decir, revisar si una variable fue declarada anteriormente, verificar divisiones por cero y como se mencionó anteriormente, comprobar que un identificador no tuviera más de 12 letras.

Finalmente, no se logró desarrollar la fase de generación de código, la razón de esto fue la complejidad de la misma, pues al tener una gramática más grande y con tantas posibilidades es difícil implementar una solución que trabaje sin errores, sumado a lo anterior, el tiempo disponible también influyó pues no era seguro que

se lograra desarrollar exitosamente esta fase, de ahí que se tomara la decisión de no hacerla.

Referencias

Tomassetti, F. (s.f.). *Listeners and Visitors*. Strumenta. Recuperado de <https://tomassetti.me/listeners-and-visitors/>

Stack Overflow. (2015). *How should I limit length of an ID token in ANTLR?* Stack Overflow. Recuperado de <https://stackoverflow.com/questions/33669709/how-should-i-limit-length-of-an-id-token-in-antlr>.