



Instituto Tecnológico de Costa Rica

Ingeniería en Computación

Compiladores e Intérpretes

Proyecto II | Análisis Sintáctico

Integrantes:

Casey Baeza Castrillo– Carné: 2022437750

Esteban Rojas Hidalgo – Carné: 2022078483

Profesor:

Allan Rodríguez Dávila

Verano 2023

Índice

| | |
|--------------------------|----|
| Manual de usuario | 3 |
| Pruebas de funcionalidad | 6 |
| Descripción del problema | 10 |
| Librerías usadas | 11 |
| Análisis de resultados | 12 |
| Bitácora | 13 |

Manual de usuario

- Requisitos Previos

Java Development Kit (JDK) instalado.

Apache Maven instalado y configurado correctamente.

JFlex y Cup instalado correctamente:

JFlex: <https://www.jflex.de/download.html>

Cup: <http://www2.cs.tum.edu/projects/cup/install.php>

Instrucciones:

1. Abrir proyecto

- a. Opción 1: Clonar repositorio

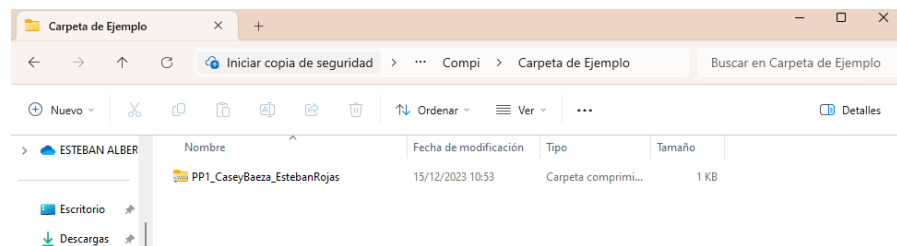
```
git clone https://github.com/cbaeza16/lexer.git
```

```
cd lexer
```

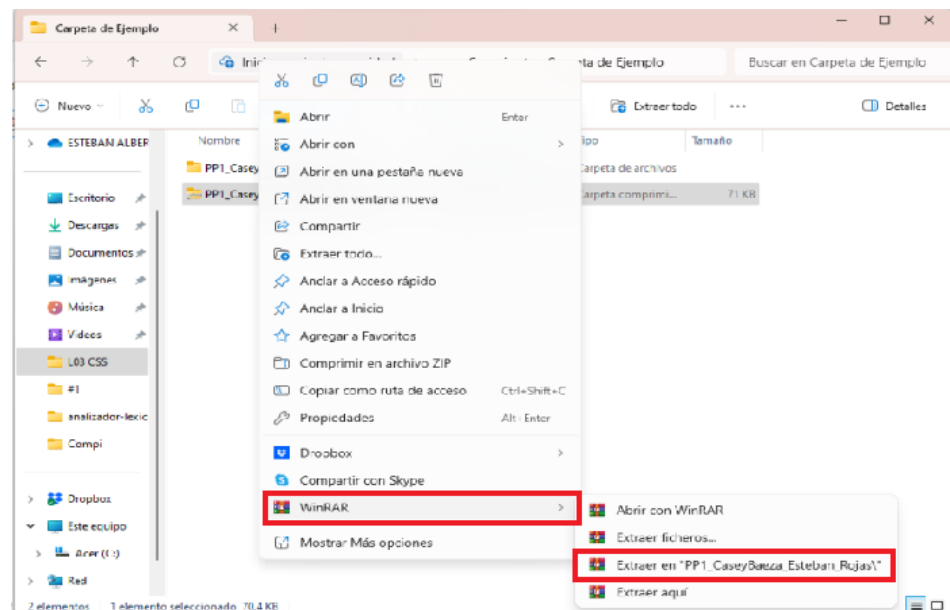
```
cd analizador-lexico
```

- b. Opción 2: Descargar .zip

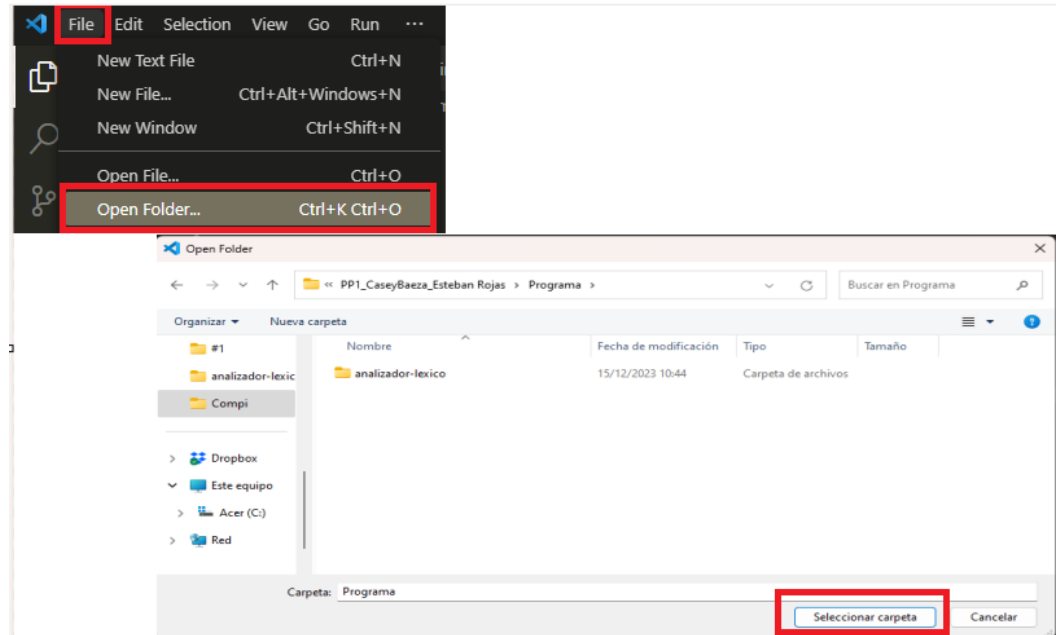
- i. Copiar el .zip en la carpeta destino deseada



- ii. Extraer los archivos en la carpeta seleccionada



- iii. Abrir el proyecto “analizador-sintactico” ("PP2_CaseyBaeza_Esteban Rojas\Programa\analizador-lexico") desde el ambiente de su preferencia



2. Compilar proyecto

`mvn clean package`

3. Ejecución del proyecto

- a. Ejecutar el programa

`mvn exec:java`

Al usar este comando se ejecuta el programa (método main) en el cual se escanea el archivo fuente “text.txt”. En caso de querer utilizar otro archivo fuente, agregar archivo fuente bajo la ruta relativa “src/main/” y cambiar el nombre del archivo en “fileName” en método main (“src/main/java/com/p1/Main.java”).

- b. En caso de querer generar Analizador Léxico:

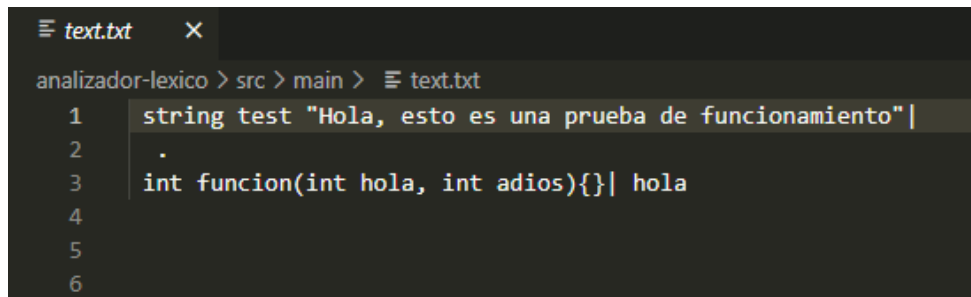
`mvn jflex:generate`

- c. En caso de querer generar Analizador Sintáctico:

`mvn cup:generate`

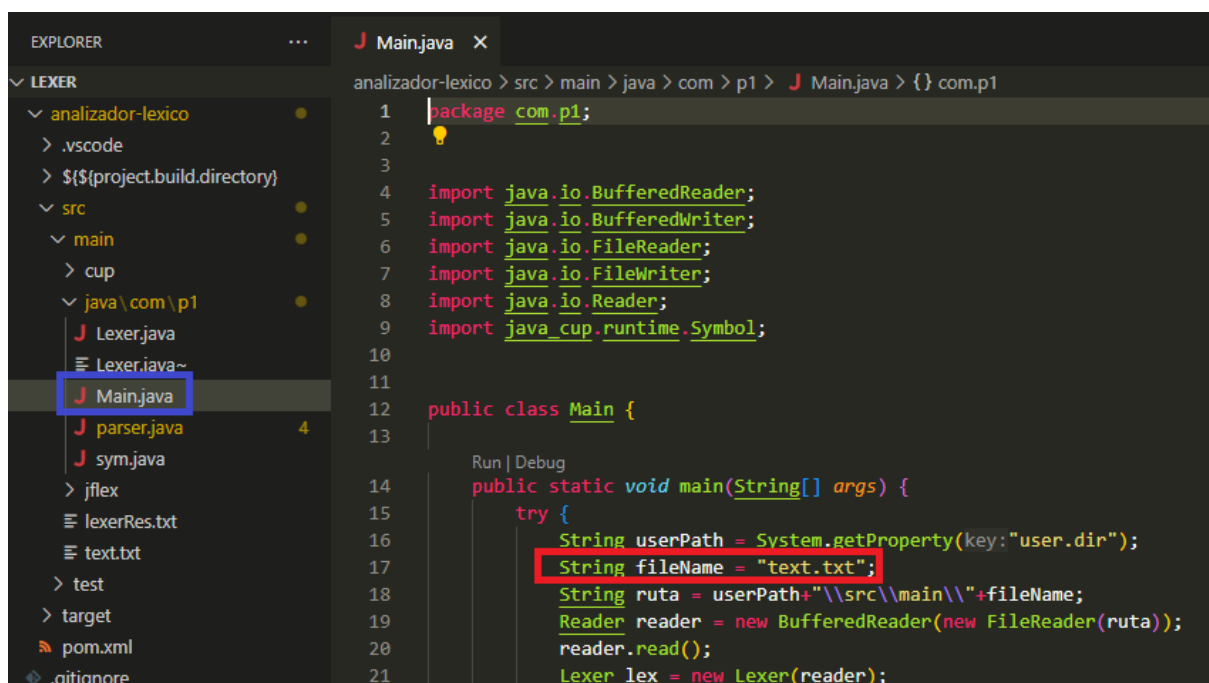
Pasos para su uso

Localizar o crear el archivo que se quiere analizar. Para este ejemplo se utiliza un archivo “text.txt”:



```
analizador-lexico > src > main > text.txt
1  string test "Hola, esto es una prueba de funcionamiento"|
2  .
3  int funcion(int hola, int adios){}| hola
4
5
6
```

Dentro del proyecto, en la clase Main.java, incluir el nombre del archivo a analizar.



```
EXPLORER
└─ LEXER
   └─ analizador-lexico
      └─ .vscode
      └─ ${project.build.directory}
      └─ src
         └─ main
            └─ cup
            └─ java\com\p1
               └─ Lexer.java
               └─ Lexer.java~
               └─ Main.java
               └─ parser.java
               └─ sym.java
            └─ jflex
            └─ lexerRes.txt
            └─ text.txt
            └─ test
            └─ target
            └─ pom.xml
            └─ .gitignore

Main.java
analizador-lexico > src > main > java > com > p1 > Main.java > {} com.p1
1  package com.p1;
2
3
4  import java.io.BufferedReader;
5  import java.io.BufferedWriter;
6  import java.io.FileReader;
7  import java.io.FileWriter;
8  import java.io.Reader;
9  import java_cup.runtime.Symbol;
10
11
12 public class Main {
13
14 Run | Debug
15 public static void main(String[] args) {
16     try {
17         String userPath = System.getProperty(key:"user.dir");
18         String fileName = "text.txt";
19         String ruta = userPath+"\\src\\main\\"+fileName;
20         Reader reader = new BufferedReader(new FileReader(ruta));
21         reader.read();
22         Lexer lex = new Lexer(reader);
```

Ya está listo para correr el programa. En la carpeta del proyecto se generará un archivo “lexerRex.txt” con los tokens encontrados y su respectiva línea y columna.

Pruebas de funcionalidad

Documento a analizar:

La siguiente imagen muestra un fragmento del documento “test.txt” utilizado para hacer pruebas:

```
src > main > ͡ text.txt
1  @Pruebas funciones
2  function int hola () {
3      break |
4      test() |
5      return tom |
6      return 29 |
7      return false |
8  }
9
10 @ERROR
11 hola
12
13 function boolean test (){
14     @Prueba de operaciones logicas
15     @variables or boolean expresion and/or ... (no se permite el uso de parentesis, solo para expresiones aritmeticas)
16     @cada operacion booleana debe estar entre {}
17     local int test1 <= {3 < 4} |
18     local int test1 <= {4 => 18.5} |
19     local int test1 <= {8 != var # 2 == 4} |
20     local int test1 <= {var # var1 != 4} |
21     local int test1 <= {test2 # 28 > 6} |
22     local int test1 <= {test2 ^ 3 < 4} |
23     local int test1 <= { 3 > 4 # test1} |
24     local int test1 <= {test # (3 + (i**2 - 8)) > 8 ^ !test2 } # {var == 3} ^ !{var5 != (3**3 * 8)} |
25     local int test1 <= {(8 + funcionArit()) < 80 ^ false} # {funcionBool() # true}|
26     local int test1 <= {true # true ^ false # {false # true} ^ false # true # !{8 < z # {true} # (8+z**2) > t}} |
27     @Prueba combinacion artmetica unaria y binaria, y logica
28     local int test1 <= { 1 == (test + ++prueba) ^ tomato < 4 # 25 != 4} |
29 }
30
31 function int sum (int num1, int num2) {
32     @Prueba de aritmetica
33     @NOTA: Siempre entre parentesis ( )
34     @TODO: toma el 8+9 como float
35     local int test1 <= (4 + 2 + 9 - - 3 / 5) |
36     local int test2 <= (6)|
37     local int test1 <= ((7 + 8 - 4)) |
38     local int test1 <= (4 - (8 + 5.26 * 3)) |
39     local int test1 <= (3 + 8 - 9.5 + (4 + 8) + 8 - (9 * 10)) |
40     local int test1 <= (((4 / 3 + 19)) + 8 - 9**8 + (8 / 9) - 3) |
41
42     local int test1 <= ((num2 + 1) - 3 / num2 * (num1 + num2)) |
43     local int test1 <= (89**9 + funcionArit() * 89) |
44 }
```

Salida al correr el programa:

La respuesta de correr la función main se divide en 3 partes principales

El analizador lexico que devuelve una descripción breve de cada token encontrado

```
PROBLEMS 9 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Token: 27, Lexema: 2, Linea: 23, Columna: 40
Token: 3, Lexema: -, Linea: 23, Columna: 42
Token: 27, Lexema: 8, Linea: 23, Columna: 44
Token: 32, Lexema: ), Linea: 23, Columna: 45
Token: 32, Lexema: ), Linea: 23, Columna: 46
Token: 13, Lexema: >, Linea: 23, Columna: 48
Token: 27, Lexema: 8, Linea: 23, Columna: 50
Token: 17, Lexema: ^, Linea: 23, Columna: 52
Token: 18, Lexema: !, Linea: 23, Columna: 54
Token: 19, Lexema: test2, Linea: 23, Columna: 55
Token: 36, Lexema: }, Linea: 23, Columna: 61
Token: 16, Lexema: #, Linea: 23, Columna: 63
Token: 35, Lexema: {, Linea: 23, Columna: 65
Token: 19, Lexema: var, Linea: 23, Columna: 66
Token: 10, Lexema: ==, Linea: 23, Columna: 70
Token: 27, Lexema: 3, Linea: 23, Columna: 73
Token: 36, Lexema: }, Linea: 23, Columna: 74
Token: 17, Lexema: ^, Linea: 23, Columna: 76
Token: 18, Lexema: !, Linea: 23, Columna: 78
Token: 35, Lexema: {, Linea: 23, Columna: 79
Token: 19, Lexema: var5, Linea: 23, Columna: 80
Token: 15, Lexema: !=, Linea: 23, Columna: 85
Token: 31, Lexema: (, Linea: 23, Columna: 88
Token: 27, Lexema: 3, Linea: 23, Columna: 89
Token: 7, Lexema: **, Linea: 23, Columna: 90
Token: 27, Lexema: 3, Linea: 23, Columna: 92
Token: 4, Lexema: *, Linea: 23, Columna: 94
Token: 27, Lexema: 8, Linea: 23, Columna: 96
Token: 32, Lexema: ), Linea: 23, Columna: 97
Token: 36, Lexema: }, Linea: 23, Columna: 98
Token: 52, Lexema: |, Linea: 23, Columna: 100
Token: 38, Lexema: local, Linea: 24, Columna: 4
Token: 21, Lexema: int, Linea: 24, Columna: 10
Token: 19, Lexema: test1, Linea: 24, Columna: 14
Token: 48, Lexema: <=, Linea: 24, Columna: 20
Token: 35, Lexema: {, Linea: 24, Columna: 23
Token: 31, Lexema: (, Linea: 24, Columna: 24
Token: 27, Lexema: 8, Linea: 24, Columna: 25
Token: 2, Lexema: +, Linea: 24, Columna: 27
Token: 19, Lexema: funcionArit, Linea: 24, Columna: 29
Token: 31, Lexema: (, Linea: 24, Columna: 40
Token: 32, Lexema: ), Linea: 24, Columna: 41
Token: 32, Lexema: ), Linea: 24, Columna: 42
```

Luego presenta un análisis sintáctico imprimiendo errores e imprimiendo los tokens de las operaciones aritméticas y logicas

```
Cantidad de lexemas encontrados: 718

-----Fin Fase Lexica-----

-----Inicio Fase Sintactica-----

Inicio de funcion hola
test
Error Sintáctico: "hola" Linea: 11 Columna: 1

Inicio de funcion test
{
3
<
4
}
{
4
=>
18.5
}
{
8
!=
var
#
2
==
4
}
{
var
#
var1
!=
4
}
{
test2
```

Por último genera impresión de las tablas de símbolos generadas con sus respectivos tokens y tipos


```
***** Cargando Tabla de Simbolos *****
```

```
Tabla de simbolo: test
```

```
Valores :
```

```
Tipo:function - Retorna:boolean
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tabla de simbolo: holaa
```

```
Valores :
```

```
Tipo:function - Retorna:int
```

```
Tipo:Param - ID:var2:int
```

```
Tipo:Param - ID:var3:boolean
```

```
Tipo:Param - ID:var4:int
```

```
Tipo:Local - ID:i:int
```

```
Tabla de simbolo: sum
```

```
Valores :
```

```
Tipo:function - Retorna:int
```

```
Tipo:Param - ID:num1:int
```

```
Tipo:Param - ID:num2:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test2:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

```
Tipo:Local - ID:test1:int
```

Se muestra un último mensaje final según la cantidad de errores encontrados por el analizador sintáctico

```
-----Fin Fase Sintactica-----
```

```
[Done] El archivo puede ser leído con éxito
```

```
-----Fin Fase Sintactica-----
```

```
[Warning] El archivo presenta 2 errores sintacticos
```

```
Error Sintáctico: "hola" Línea: 11 Columna: 1
```

```
Error Sintáctico: "wuju" Línea: 31 Columna: 1
```

Documento generado:

Se genera un documento “lexerRes.txt” con el registro de los lexemas encontrados en la parte del analizador léxico

```
src > main > ≡ lexerRes.txt
1 Token: 39, Lexema: function, Línea: 1, Columna: 0
2 Token: 21, Lexema: int, Línea: 1, Columna: 9
3 Token: 19, Lexema: hola, Línea: 1, Columna: 13
4 Token: 31, Lexema: (, Línea: 1, Columna: 18
5 Token: 32, Lexema: ), Línea: 1, Columna: 19
6 Token: 35, Lexema: {, Línea: 1, Columna: 21
7 Token: 47, Lexema: break, Línea: 2, Columna: 4
8 Token: 52, Lexema: |, Línea: 2, Columna: 10
9 Token: 19, Lexema: test, Línea: 3, Columna: 4
10 Token: 31, Lexema: (, Línea: 3, Columna: 8
11 Token: 32, Lexema: ), Línea: 3, Columna: 9
12 Token: 52, Lexema: |, Línea: 3, Columna: 11
13 Token: 46, Lexema: return, Línea: 4, Columna: 4
14 Token: 19, Lexema: tom, Línea: 4, Columna: 11
15 Token: 52, Lexema: |, Línea: 4, Columna: 15
16 Token: 46, Lexema: return, Línea: 5, Columna: 4
17 Token: 27, Lexema: 29, Línea: 5, Columna: 11
18 Token: 52, Lexema: |, Línea: 5, Columna: 14
19 Token: 46, Lexema: return, Línea: 6, Columna: 4
20 Token: 25, Lexema: false, Línea: 6, Columna: 11
21 Token: 52, Lexema: |, Línea: 6, Columna: 17
22 Token: 36, Lexema: }, Línea: 7, Columna: 0
23 Token: 19, Lexema: hola, Línea: 10, Columna: 0
24 Token: 39, Lexema: function, Línea: 12, Columna: 0
25 Token: 20, Lexema: boolean, Línea: 12, Columna: 9
26 Token: 19, Lexema: test, Línea: 12, Columna: 17
27 Token: 31, Lexema: (, Línea: 12, Columna: 22
28 Token: 32, Lexema: ), Línea: 12, Columna: 23
29 Token: 35, Lexema: {, Línea: 12, Columna: 24
30 Token: 38, Lexema: local, Línea: 16, Columna: 4
```

Descripción del problema

Para este proyecto se debe diseñar un lenguaje imperativo especializado para la configuración de chips en sistemas empujados, es fundamental que sea un lenguaje ligero y potente ya que es una industria en constante evolución, donde los chips requieren configuraciones cada vez más precisas y complejas.

El enfoque principal se basa en la fase de Análisis Sintáctico, para una gramática en específico y el analizador léxico desarrollado en el proyecto 1. Para esto se debe desarrollar

un parser utilizando la herramienta Cup, y así realizar la comprobación sintáctica y manejo-recuperación de errores.

Diseño del programa

En base a la estructura del proyecto, se utiliza una estructura Maven, para así tener directorios bien organizados y claramente definidos para código fuente, recursos, archivos generados, etc.

En cuanto a la implementación del Analizador Sintactico, se utiliza Cup para generar el analizador sintáctico basado en las reglas definidas en la gramática del lenguaje.

Según la estructura e implementación del programa, es fundamental la configuración de plugins, es importante tener configurados los plugins de JFlex y CUP en el archivo pom.xml, especificando las rutas a los archivos de definición de la gramática y las reglas léxicas, y las rutas de salida donde se genera automáticamente los analizadores léxicos y sintácticos al compilar el proyecto.

Librerías usadas

Se desarrolla un scanner utilizando la herramienta JFlex utilizando la opción %cup, con el fin que los tokens retornados de tipo Symbol utilizando la clase sym. Por esto, se utilizan las librerías de JFlex y Java CUP que contienen las clases y métodos necesarios para el funcionamiento de los analizadores léxicos y sintácticos.

Cabe mencionar que debido a la estructura utilizada es necesario incluir ciertas dependencias y plugins, a pesar de no ser librerías como tal, para generar automáticamente los analizadores léxicos y sintácticos al compilar el proyecto.

Uno de los plugins utilizados es jflex-maven-plugin, que permite integrar la generación de analizadores léxicos basados en archivos de definición de gramática con el ciclo de vida de construcción de Maven. Asimismo, se establece la ubicación de los archivos de definición de gramática JFlex y se especifica el directorio de salida para el código generado. En este caso, el código Java resultante se coloca en src/main/java.

```
<!-- plugin JFlex -->
<plugin>
  <groupId>de.jflex</groupId>
  <artifactId>jflex-maven-plugin</artifactId>
  <version>1.9.1</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

        <configuration>
        <outputDirectory>src/main/java</outputDirectory>
        <lexDefinitions>
            <lexDefinition>src/main/jflex</lexDefinition>
        </lexDefinitions>
        </configuration>
    </execution>
</executions>
</plugin>

```

Otro de los plugins utilizados es el cup-maven-plugin, junto con su dependencia asociada, utilizado para generar analizadores sintácticos basados en CUP. Además, se especifica el directorio de salida para el código generado. En este caso, el código Java resultante se coloca en src/main/java..

```

<!-- plugin Cup -->
<plugin>
    <groupId>com.github.vbmacher</groupId>
    <artifactId>cup-maven-plugin</artifactId>
    <version>11b-20160615-2</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <outputDirectory>src/main/java</outputDirectory>
    </configuration>
</plugin>

<!-- Dependencia Cup -->
<dependencies>
    <dependency>
        <groupId>com.github.vbmacher</groupId>
        <artifactId>java-cup-runtime</artifactId>
        <version>11b-20160615-2</version>
    </dependency>

```

Análisis de resultados

El siguiente checklist muestra los objetivos alcanzados y no alcanzados:

- ☒ Analizar sintácticamente la creación de funciones, y dentro de ellas, estructuras de control, bloques de código ({}) y sentencias de código.

- ☒ Analizar sintácticamente los tipos de variables enteras, flotantes, booleanas, caracteres, cadenas de caracteres (string) y arreglo estático.
- ☒ Analizar sintácticamente la creación de arreglos de tipo entero o char. Además, de obtener y modificar sus elementos, y ser utilizados en expresiones.
- ☒ Analizar sintácticamente las sentencias para creación de variables, creación y asignación de expresiones y asignación de expresiones a variables, y algunos casos, sólo expresiones sin asignación.
- ☒ Analizar sintácticamente operadores y operandos, respetando precedencia (usual matemática) y permitiendo el uso de paréntesis.
- ☒ Analizar sintácticamente expresiones aritméticas binarias de suma (+), resta (-), división (/) –entera o decimal según el tipo--, multiplicación (*), módulo (~) y potencia (**).
- ☒ Analizar sintácticamente expresiones aritméticas unarias de negativo (-), ++, -- , antes del operando.
- ☒ Analizar sintácticamente expresiones relacionales (sobre enteros y flotantes) de menor, menor o igual, mayor, mayor o igual, igual y diferente. En la que los operadores igual y diferente permiten adicionalmente tipo booleano.
- ☒ Analizar sintácticamente expresiones lógicas de conjunción (^), disyunción (#) y negación (!).
- ☒ Analizar sintácticamente sentencias de código para las diferentes expresiones mencionadas anteriormente y su combinación.
- ☒ Analizar sintácticamente el uso de tipos y la combinación de expresiones aritméticas (binarias y unarias), relacionales y lógicas, según las reglas gramaticales, aritméticas, relacionales y lógicas del Paradigma Imperativo.
- ☒ Analizar sintácticamente las estructuras de control if-[elif]-[else], do-until y for, además, permitir return y break.
- ☒ Analizar sintácticamente las funciones de leer y escribir en la salida estándar..
- ☒ Analizar sintácticamente la creación y utilización de funciones, definición clásica..
- ☒ Analizar sintácticamente un único procedimiento inicial main.
- ☒ Debe permitir comentarios de una línea (@) o múltiples líneas (/ _ _/).

Bitácora