**Universidad Internacional de La Rioja**

Nombre:

Arnaldo Quintero Segura

Asignatura:

Procesadores de Lenguajes

Profesor:

Dr. Ing. Fernando López Hernández

Lab 02 - Analizador sintáctico

Fecha:

03 de mayo de 2020

ESPAÑA - LA RIOJA

[Lab 02 - Analizador sintáctico 1](#_Toc39424437)

[Problemas encontrados: 1](#_Toc39424438)

[Parseo de ficheros: 1](#_Toc39424439)

[Bloques anidados: 2](#_Toc39424440)

[Consideraciones y desiciones de diseño 2](#_Toc39424441)

[Ejecución del código 3](#_Toc39424442)

[Compilación: 3](#_Toc39424443)

[Parseo de caso de prueba: 4](#_Toc39424444)

[Bibliografía 9](#_Toc39424445)

# Lab 02 - Analizador sintáctico

Para la realización de este analizador léxico, se ha utilizado la herramienta JFlex, junto con la herramienta CUP, siguiendo los pasos expuestos en las indicaciones.

## Problemas encontrados:

Como principal problema, se ha encontrado el parseo de los ficheros cuando existen muchas cadenas vacías. Ya que el lenguaje debe tomar en cuenta los saltos de línea como delimitadores. Fue un poco difícil lograr que la gramática aceptara ficheros vacíos, ficheros con instrucciones, ficheros con líneas vacías (solo contiene el salto de línea), y la mezcla de todos estos.

Además de la posibilidad de tener bloques anidados de grupos de control, tales como definiciones de funciones con bloques "for" dentro, que dentro incluyen bloques "if" y demás. Veamos a continuación cómo fueron resueltos estos problemas:

### Parseo de ficheros:

Para la resolución de este problema, simplemente se plantea esta solución gramatical:

(1)program ::= actions

| new\_lines

;

(2)actions ::= new\_lines statement statements

;

(3)new\_lines ::= LINE\_END new\_lines

|

;

(4)statements ::= LINE\_END new\_lines statement statements

| new\_lines

;

Esto nos permite, que el programa pueda ser vacío, o simplemente lleno de líneas vacías. Si se utiliza la segunda expansión de la regla (1). De lo contrario, si se sigue por la primera expansión, significa que tendremos al menos una línea de código. Bien sea terminada con el fin de línea o con el fin del fichero.

En este caso, se debe prever que antes de esta línea de código, existan o no líneas vacías. Y luego, una acción puede estar seguida de una o más acciones, siempre teniendo un fin de línea antes de la nueva acción. Esto se logra con la ayuda de la regla (4).

### Bloques anidados:

Para la resolución de este problema, fue necesario cambiar el fichero de análisis Léxico, siguiendo lo hecho por (Syrion89, 2016) en el cuál se ha visto que se posee una pila dentro del lexer, junto con un par de variables de tipo entero, que controlan si el código se encuentra actualmente dentro de un bloque anidado o no.

Haciendo así que, al analizar un bloque anidado, se pueda ver un token de "TAB" al iniciar un bloque indentado y se crea el token "DEDENT", tal como nos dice la gramática original de Python (The Python Software Foundation, s.f. a). El cual nos demuestra que el bloque de indentación ha terminado.

Con respecto al código de (Syrion89), el código implementado ha sido modificado para que sólo acepte tabulaciones de un carácter de tabulador. Mientras que el código original acepta como tabulación uno o más tabuladores.

Siguiendo esto, la gramática luego es escrita de manera que los bloques compuestos indentados comiencen con un token "TAB" y terminen con un "DEDENT". Permitiendo así un parseo de tokens recursivo dentro del cuerpo anidado. En el cual se aceptan cualquier tipo de acción o "statement".

Esto mismo nos genera una consecuencia, la cual es que no se permiten líneas fuera de la indentación actual. Sólo al comienzo del bloque. De manera que antes de comenzar un bloque indentado puedan existir líneas vacías o comentarios sin indentar, pero al comenzar la primera acción del bloque, todas las líneas deben seguir dicha indentación. Lo cual significa que los comentarios deben cumplir también la indentación.

## Consideraciones y decisiones de diseño

En cuanto a comprobaciones generales, se puede ver que la instrucción "return" es manejada como una acción normal. A pesar de que no debe ser posible ejecutarla fuera de una función. Pero luego de ver las referencias señaladas. Se asume que el trabajo de revisar que este código este dentro de una función pertenecerá a la siguiente fase del desarrollo.

Ya que una función puede contener cualquier tipo de bloque interno de código, y al privar estos bloques de dicha instrucción se generaría una duplicación de código y reglas gramaticales masivas.

Como decisión de diseño, esta versión de miniPython soporta la encadenación de operadores, tanto aritméticos como booleanos.

Pero a su vez, se decide no aceptar suma de caracteres de texto, ya que no fue tomado en cuenta a momento de escribir la gramática y no se muestra como un requerimiento en las especificaciones. Y de ser necesario para la siguiente fase, no supondría una gran dificultad agregar esta función

## Ejecución del código

Para la ejecución del código, se entrega junto a este documento, un comprimido ZIP que contiene los recursos necesarios para ejecutar dicho código, así como el código fuente que son los ficheros: "Analyzer.java", "lexicon.flex" y "syntax.cup"; así como dos ficheros de utilidades: "clean" y "test".

El fichero "clean" es un fichero bash ejecutable que se encarga de limpiar los posibles ficheros residuales creados luego de compilar y ejecutar el código.

Por otro lado, el fichero "test", se encarga degenerar utilizando Java CUP y JFlex los ficheros "Lexer.java" y "parser.java", haciendo uso de los binarios de CUP y JFlex incluidos en la carpeta "Resources". Además de esto, se compilan estos dos ficheros generados, junto con el "Analyzer.java" Y luego se procede a ejecutar uno a uno todos los ficheros dentro de la carpeta "Examples", los cuales describen distintos casos de prueba para el analizador.

Veamos paso a paso el proceso de compilación y un ejemplo de ejecución del fichero "00.minipy" el cual contiene el compilado de los trozos de código que se encontraban en la especificación del lenguaje, pertenecientes al enunciado del Laboratorio 1.

### Compilación:

java -jar ./Resources/java-cup-11b.jar syntax.cup

java -jar ./Resources/jflex-full-1.8.1.jar ./lexicon.flex

javac -cp .:Resources/jflex-full-1.8.1.jar ./Lexer.java

javac -cp .:Resources/jflex-full-1.8.1.jar ./parser.java

javac -cp .:Resources/jflex-full-1.8.1.jar ./Analyzer.java

Al ejecutar este código, se obtiene el siguiente resultado:

A screenshot of a cell phone

Description automatically generated

El cual nos muestra que todas las compilaciones han sido correctas, y nos da información respecto al parser generado.

### Parseo de caso de prueba:

Como caso de prueba, fichero "00.minipy" contiene el siguiente código:

# this is the first comment

spam = 1 # and this is the second comment

height = 5 \* 9

x = 5 \*\* 2 # 5 squared

tax = 12.5 / 100

s = 'First line'

s2 = "Second line"

meses = ('Enero', 'Febrero', 'Marzo') # tupla

impares = [1,3,5] # lista

params = {'Width':30, "Height":20}

print height

print(height)

if x < 0:

x=0

else:

print 'More'

endif

for m in meses:

print m

endfor

for x in impares:

if x in impares:

print(x)

endif

endfor

def suma(x):

ret = 0

for n in x:

ret = ret + n

endfor

return ret

enddef

x = [2, 4, 7, 12, 3]

total = suma(x)

Y al ejecutar el "Analyzer" con este fichero como entrada, utilizando el comando:

java -cp .:Resources/jflex-full-1.8.1.jar Analyzer ./Ejemplos/00.minipy

Se obtiene entonces el siguiente resultado, el cual muestra las salidas del lexer, y al final el indicador de que la cadena ha sido parseada correctamente:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

# Bibliografía

Foundation, T. P. S. (a). *10. full grammar specification — python 3.8.3rc1 documentation* docs.python.org. Recuperado de <https://docs.python.org/3/reference/grammar.html>

Foundation, T. P. S. (b). *8. compound statements — python 3.8.3rc1 documentation* docs.python.org. Recuperado de <https://docs.python.org/3/reference/compound_stmts.html>

Foundation, T. P. S. (c). *9. top-level components — python 3.8.3rc1 documentation* docs.python.org. Recuperado de <https://docs.python.org/3/reference/toplevel_components.html>

Syrion89. (2016). *Syrion89/LexerParserPython* GitHub. Recuperado de <https://github.com/Syrion89/LexerParserPython>