# Analizador Léxico y Sintáctico para Python Implementación con Flex/Bison

Arregoces, Gonzalez, Sanchez, Oviedo
*Laboratorio de Compiladores*
Universidad del Norte
Barranquilla, Colombia

**Resumen**

Este documento presenta la implementación de un analizador léxico y sintáctico completo para un subconjunto del lenguaje Python, desarrollado como parte del Laboratorio 2 de la asignatura Compiladores. El sistema permite validar la estructura sintáctica de programas en Python simplificado, identificar errores léxicos y sintácticos, y generar archivos de salida con reportes detallados de los resultados del análisis. El proyecto fue implementado en C utilizando Flex y Bison, y se ejecuta tanto localmente en Linux como en contenedores Docker mediante un Makefile unificado.

**Index Terms**

Compiladores, Análisis Léxico, Análisis Sintáctico, Flex, Bison, Python, Docker

## I. Introducción

El proceso de compilación de un programa comienza con las fases de análisis léxico y sintáctico. El análisis léxico convierte el código fuente en una secuencia de tokens, mientras que el análisis sintáctico valida la estructura gramatical del programa conforme a las reglas del lenguaje.

Este laboratorio integra ambos procesos, construyendo un analizador léxico y sintáctico capaz de procesar un subconjunto del lenguaje Python. El sistema detecta errores en la escritura de programas y genera reportes de análisis detallados, permitiendo la validación automática de código fuente.

## II. Descripción del Sistema

### *II-A. Analizador Léxico*

El analizador léxico, desarrollado en Flex, identifica los siguientes elementos:

- Más de 19 palabras reservadas (`if`, `else`, `def`, `print`, `import`, etc.)
- Tipos numéricos: enteros, reales, largos, imaginarios y notación científica
- Operadores aritméticos, de comparación, lógicos, bit a bit y de asignación
- Identificadores válidos, cadenas, delimitadores y comentarios
- Detección y conteo de errores léxicos

### *II-B. Analizador Sintáctico*

El analizador sintáctico, implementado en Bison, verifica la corrección gramatical de los programas en Python simplificado. Entre las estructuras soportadas se incluyen:

- Asignaciones simples y múltiples
- Definición de funciones con parámetros y retorno
- Condicionales: `if`, `elif`, `else`
- Ciclos: `for`, `while`
- Sentencias de control: `break`, `continue`, `pass`
- Instrucciones `print`, `import`, y funciones `range()`, `len()`

El sistema genera archivos `.txt` con los resultados del análisis:

- `<archivo>_lexico_tokens.txt`: resultado del análisis léxico
- `<archivo>_sintactico.txt`: resultado del análisis sintáctico

## III. Requisitos del Sistema

### III-A. Ejecución local

- Ubuntu 20.04 o superior
- Flex y Bison instalados
- Compilador `gcc`

Instalación de dependencias:

```
sudo apt update
sudo apt install flex bison gcc make
```

### III-B. Ejecución con Docker

Alternativamente, el proyecto puede ejecutarse dentro de un contenedor que ya incluye todas las herramientas necesarias (Flex, Bison y GCC).

```
git clone https://github.com/flaviofuego/Lab_Analisis_Lexico-Compiladores.git
cd Lab_Analisis_Lexico-Compiladores
docker build -t analizador-lexico .
```

## IV. Ejecución del Sistema

### IV-A. Ejecución local

1. Compilar los analizadores:

```
bison -d LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.y
flex LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.l
gcc -o LAB02_Arregoces_Gonzalez_Sanchez_Oviedo \
    lex.yy.c LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.tab.c -lfl
```

2. Ejecutar el analizador con un archivo de prueba:

```
./LAB02_Arregoces_Gonzalez_Sanchez_Oviedo entradas/prueba1.py
```

3. Verificar los resultados en el directorio `salidas/`.

### IV-B. Ejecución mediante Docker y Makefile

El proyecto incluye un Makefile unificado que simplifica la compilación y ejecución:

```
# Compilar analizadores
docker run --rm -v "${PWD}:/workspace" analizador-lexico make build

# An lisis completo
docker run --rm -v "${PWD}:/workspace" analizador-lexico make completo FILE=entradas/prueba1
    .py
```

## V. Comandos del Makefile

El Makefile permite automatizar las tareas de compilación, ejecución y limpieza del proyecto. A continuación se describen algunos de los comandos más importantes:

### V-A. `make help`

Muestra una lista completa de los comandos disponibles junto con una breve descripción de cada uno. Es útil para recordar las opciones de ejecución y los parámetros que puede aceptar el sistema.

```
make help
```

### V-B. `make build`

Compila ambos analizadores (léxico y sintáctico) generando los archivos intermedios: `lex.yy.c`, `.tab.c`, `.tab.h` y el ejecutable principal.

```
make build
```

### V-C. `make sintactico`

Ejecuta únicamente el análisis sintáctico sobre el archivo especificado. Requiere el parámetro `FILE=` indicando el archivo de entrada.

```
make sintactico FILE=entradas/prueba2.py
```

### V-D. `make clean-all`

Elimina todos los archivos generados por el proceso de compilación, incluyendo los binarios, archivos intermedios y las salidas léxicas y sintácticas.

```
make clean-all
```

## VI. Estructura del Proyecto

```
Lab_Analisis_Lexico-Compiladores/
            src/
                  LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.l   # Analizador l xico (Flex)
                  LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.y   # Analizador sint ctico (Bison)
            entradas/
                  prueba1.py
                  prueba2.py
                  prueba_correcta.py
            salidas/
                  *_lexico_tokens.txt
                  *_sintactico.txt
            dist/
                  LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.tab.c
                  LAB02_Arregoces_Gonzalez_Sanchez_Oviedo.tab.h
                  LAB02_Arregoces_Gonzalez_Sanchez_Oviedo
                  lex.yy.c
            Makefile
            Dockerfile
            README.md
```

## VII. Ejemplos de Resultados

### VII-A. Salida correcta

```
Prueba con el archivo de entrada
0 errores
```

### VII-B. Salida con errores

```
Prueba con el archivo de entrada
línea 1 error
línea 3 error
línea 4 error
```

## VIII. Conclusiones

Se ha implementado un analizador léxico y sintáctico completo para un subconjunto de Python. El sistema permite detectar y reportar errores con precisión, generando salidas legibles tanto en la fase léxica como sintáctica.

El uso de herramientas como Flex y Bison facilita la definición formal de la gramática, mientras que el entorno Docker garantiza la portabilidad del sistema. El Makefile unificado simplifica la compilación, ejecución y pruebas del proyecto.

Este laboratorio constituye un paso fundamental en el proceso de construcción de un compilador completo, sirviendo como base para el análisis semántico y la generación de código intermedio.