



## **Trabajo Práctico N° 1**

### **Parseo y generación de código**

### **Analizador Léxico de Eiffel**

# Resumen

Este documento contiene una implementación mínima de la parte léxica del trabajo práctico: incluye los archivos **lexer.l** (Flex), **parser.y** (Bison), un **Makefile**, ejemplos de entrada (.e) en la carpeta **tests** y las salidas esperadas. También se incluye una guía paso a paso para compilar.

---

## Estructura propuesta de entregables

- **lexer.l** - definiciones Flex (expresiones regulares y acciones).
  - **parser.y** - Bison (archivo que coordina la lectura de tokens y genera la salida solicitada).
  - **Makefile** - reglas para generar **parser.tab.c**, **lex.yy.c** y el ejecutable **eiffel\_lex**.
  - **tests/** - ejemplos eiffel en formato .e, archivos **.expected** con el resultado esperado, y **.result** con el output del parser que luego es comparado contra el archivo **.expected**.
- 

## Cómo compilar y ejecutar (instrucciones)

**Requisitos:** flex, bison, gcc y la librería libfl (generalmente provista por flex en linux).

En la terminal, estando en la carpeta del proyecto hay que ejecutar los siguientes comandos en orden:

Para generar parser y cabecera (genera archivos **parser.tab.c** y **parser.tab.h**):

**bison -d -o parser.tab.c parser.y**

Generar lexer (lexer.yy.c):

**flex -o lex.yy.c lexer.l**

Compilar (genera **eiffel\_lex**):

en **linux**:

**gcc -o eiffel\_lex parser.tab.c lex.yy.c -lfl -Wall -g**

en **macOS**:

**gcc -o eiffel\_lex parser.tab.c lex.yy.c -Wall -g**

Ejecutar sobre un archivo de prueba:

**./eiffel\_lex tests/hello.e > tests/hello.result**

Comparar con la salida esperada:

```
diff -u tests/hello.expected tests/hello.result || true
```

#### Nota:

Si bison -d genera parser.tab.h (nombre típico), tanto lexer.l como parser.y incluyen esa cabecera. El Makefile incluido usa bison -d para crear la cabecera automáticamente.

---

## Diseño y decisiones de implementación

### Objetivo

El analizador léxico detecta y clasifica tokens básicos del lenguaje Eiffel: palabras reservadas (class, create, feature, do, end, if, then, else, from, until, loop, ...), identificadores, constantes numéricas (enteros y reales), constantes de cadena, operadores y símbolos, y comentarios.

### Principales decisiones

- **Palabras reservadas:** se reconocen todas las palabras reservadas en minúsculas.
  - **Identificadores:** patrón `[A-Za-z_][A-Za-z0-9_]*`.
  - **Números:** se reconocen enteros y reales (con parte fraccionaria opcional y exponente opcional).
  - **Cadenas:** `"([^\n]|\n)*"` (soporta escapes como `\`, `\\`, `\n`, etc.).
  - **Comentarios:** `--` hasta fin de línea (ignorados).
  - **Operadores compuestos** (`<=`, `>=`, `:=`) se colocan antes de los simples (`<`, `>`, `:`) para evitar conflictos.
  - **Salida:** Bison se usa como coordinador. El `main()` invoca `yyparse()` y se imprime una línea por token con su tipo y lexema.
- 

## Explicación de expresiones regulares (resumen)

- **Identificador:** `[A-Za-z_][A-Za-z0-9_]*`, empieza con letra o `_` y puede tener dígitos después.
- **Entero/Real:** `[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?` — reconoce 123, 3.14, 2.0E-3.

- **Cadena:** "[<sup>^</sup>"\n]|\\.)\*" — abre y cierra con ", permite escapes \.
  - **Comentario:** -- seguido de cualquier cosa hasta \n.
  - **Operadores:** patrones literales ":", "<=", ">=", "+", "-", "\*", "/", etc.
  - **Espacios:** [\t\r]+ se omiten; \n se deja para mantener número de línea con yylineno.
- 

## Código fuente Makefile

### Makefile

```
BISON=bison
FLEX=flex
CC=gcc
CFLAGS=-Wall -g
UNAME_S:= $(shell uname -s)



all: eiffel_lex

parser.tab.c parser.tab.h: parser.y
    $(BISON) -d -o parser.tab.c parser.y

lex.yy.c: lexer.l parser.tab.h
    $(FLEX) -o lex.yy.c lexer.l

eiffel_lex: parser.tab.c lex.yy.c
ifeq ($(UNAME_S), Darwin)
    $(CC) $(CFLAGS) -o eiffel_lex parser.tab.c lex.yy.c
else
    $(CC) $(CFLAGS) -o eiffel_lex parser.tab.c lex.yy.c -lfl
endif

clean:
    rm -f parser.tab.c parser.tab.h lex.yy.c eiffel_lex

test: eiffel_lex
    @for t in $(TESTS); do \
        echo "Running test $$t..."; \
        ./eiffel_lex < $$t.e > $$t.result; \
        if diff -q $$t.result $$t.expected > /dev/null; then \
            echo "  PASSED"; \
            rm -f $$t.result; \
        else \
            echo "  FAILED (see $$t.result vs $$t.expected)"; \
        fi \
    done
.PHONY: all clean test
```

---

## Tests de ejemplo

El sistema de pruebas se encuentra en el directorio `tests/` y se compone de pares de archivos para cada caso de prueba:

Archivo de Entrada (**.e**): Contiene un fragmento de código fuente en Eiffel que prueba una o varias características del lenguaje (bucles, condicionales, comentarios, etc.). Por ejemplo, `loop.e` o `if_else.e`.

Archivo de Salida Esperada (**.expected**): Contiene la secuencia exacta de tokens que el analizador léxico debería generar al procesar el archivo `.e` correspondiente. Cada línea representa un token reconocido y su valor.

## Proceso de Ejecución

Las pruebas se ejecutan con el comando **make test**. Este comando automatiza los siguientes pasos para cada caso de prueba definido en el directorio `tests/`:

Ejecución del Analizador: Se invoca el programa `eiffel_lex` y se le proporciona el archivo de código fuente (`.e`) como entrada.

Captura de la Salida: La salida generada por el analizador (la secuencia de tokens) se redirige a un archivo temporal (`.result`).

Comparación: Se utiliza la utilidad `diff` para comparar el contenido del archivo `.result` con el archivo de salida esperada (`.expected`).

```
~/UNQ/parseo-codigo-2025-tp1 git:[main]
make test
Running test tests/comments...
  ✓ PASSED
Running test tests/hello...
  ✓ PASSED
Running test tests/if_else...
  ✓ PASSED
Running test tests/logic...
  ✓ PASSED
Running test tests/loop...
  ✓ PASSED
```

---

### Verificación:

Si los archivos son idénticos, significa que el analizador léxico ha funcionado como se esperaba y la prueba se marca como PASSED. El archivo .result temporal se elimina.

Si hay diferencias, la prueba se marca como FAILED. El archivo .result se conserva para que el desarrollador pueda inspeccionar las diferencias y depurar el error.

---

## Ejemplo:

Archivo hello.e:

```
1  class HELLO_WORLD
2  create
3      make
4  feature
5      make
6      do
7          print ("Hola, Eiffel!%N")
8      end
9  end
```

---

Archivo hello.expected (Resultado esperado):

```
1    TOKEN_CLASS -> class
2    TOKEN_IDENTIFIER -> HELLO_WORLD
3    TOKEN_CREATE -> create
4    TOKEN_IDENTIFIER -> make
5    TOKEN_FEATURE -> feature
6    TOKEN_IDENTIFIER -> make
7    TOKEN_DO -> do
8    TOKEN_IDENTIFIER -> print
9    TOKEN_LPAREN -> (
10   TOKEN_STRING -> "Hola, Eiffel!%N"
11   TOKEN_RPAREN -> )
12   TOKEN_END -> end
13   TOKEN_END -> end
14
```

---

## Nota sobre Eiffel (reseña corta)

**Eiffel** es un lenguaje de programación orientado a objetos creado a mediados de los años 80 por **Bertrand Meyer**, entonces investigador y más tarde profesor en el **ETH Zürich**. Su diseño nació con un objetivo académico y práctico: mejorar la **calidad del software** aplicando principios de ingeniería rigurosos dentro del propio lenguaje.

La gran innovación de Eiffel es el concepto de **Design by Contract (DbC)**, que integra en el código contratos formales (precondiciones, postcondiciones e invariantes) para garantizar que los módulos colaboren de forma segura y confiable. Además, Eiffel incorpora herencia múltiple



con control, tipado estático fuerte y generics, siempre priorizando la legibilidad y la mantenibilidad.

El ecosistema principal es **EiffelStudio**, un entorno completo de desarrollo que combina compilador, depurador y herramientas de análisis. Aunque su uso industrial fue limitado frente a lenguajes más populares, Eiffel tuvo gran impacto en la enseñanza de ingeniería del software y dejó un legado fuerte: la idea de contratos influyó en Java, C#, Ada y otras plataformas.

En síntesis, Eiffel es más que un lenguaje: representa una **filosofía de desarrollo confiable** originada en el ámbito universitario, que todavía inspira prácticas modernas de diseño y verificación de software.

---