

# **Pontificia Universidad Católica Madre y Maestra**



**CSTI-1880-6812 - Procesamiento de Lenguaje Compilador**

**Tema:**

Procesamiento Lenguaje y Compilación

**Integrantes:**

1014-3611 Starlin Frías Luna

**Profesor:**

Juan Felipe

**Santiago de los Caballeros, República Dominicana**

**Domingo 23 de Febrero del 2025**

# Introducción

Este documento detalla la implementación y análisis de tres ejercicios utilizando Flex (Fast Lexical Analyzer Generator), una herramienta para generar analizadores léxicos. Los ejercicios abarcan desde un reconocedor simple de palabras clave hasta un analizador léxico completo para un subconjunto del lenguaje Pascal.

## Descripción General de Flex

Flex es una herramienta que genera analizadores léxicos basados en expresiones regulares. Un analizador léxico lee un flujo de caracteres de entrada y los agrupa en secuencias significativas llamadas tokens. Flex lee un archivo de especificaciones que contiene patrones y acciones, generando código C que implementa un analizador léxico basado en estas especificaciones.

## Ejercicio 1: Reconocedor Simple de Palabras Clave

### Implementación

El primer ejercicio implementa un reconocedor simple que identifica palabras clave específicas. El código fuente (simple.flex) está estructurado de la siguiente manera:

```
%{
#include <stdio.h>
%}

%%
if      printf("KEYWORD: IF\n");
then    printf("KEYWORD: THEN\n");
else    printf("KEYWORD: ELSE\n");
for      printf("KEYWORD: FOR\n");
begin   printf("KEYWORD: BEGIN\n");
end      printf("KEYWORD: END\n");
[0-9]+   printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]* printf("WORD\n");
[ \t\n] ; /* Ignorar espacios en blanco */
.        printf("UNRECOGNIZED\n");
%%

int main() {
    yylex();
    return 0;
}
```

## Archivo de Prueba (test\_simple.txt)

```
if x then
begin
    for i
    if test then
        something
    else
        other
end
```

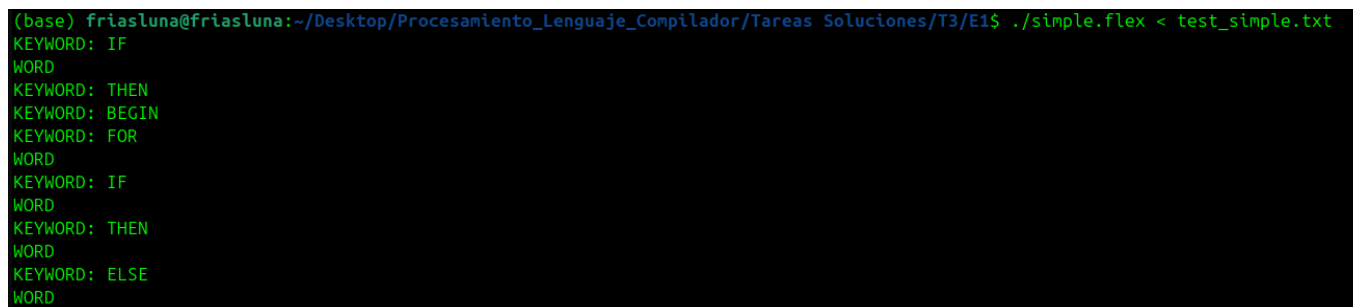
## Instrucciones de Compilación y Ejecución

```
flex simple.flex
gcc -o simple lex.yy.c -ll
./simple < test_simple.txt
```

## Salida Obtenida

```
KEYWORD: IF
WORD
KEYWORD: THEN
KEYWORD: BEGIN
KEYWORD: FOR
WORD
KEYWORD: IF
WORD
KEYWORD: THEN
WORD
KEYWORD: ELSE
WORD
KEYWORD: END
```

## Imagen:



```
(base) friasluna@friasluna:~/Desktop/Procesamiento_Lenguaje_Compilador/Tareas Soluciones/T3/E1$ ./simple.flex < test_simple.txt
KEYWORD: IF
WORD
KEYWORD: THEN
KEYWORD: BEGIN
KEYWORD: FOR
WORD
KEYWORD: IF
WORD
KEYWORD: THEN
WORD
KEYWORD: ELSE
WORD
```

## Análisis de Resultados

El analizador identifica correctamente las palabras clave definidas (if, then, else, for, begin, end) y las distingue de otras palabras. Los espacios en blanco son ignorados según lo especificado, y el programa mantiene el orden correcto de los tokens en la entrada.

## Ejercicio 2: Contador de Palabras Modificado

### Implementación

Este ejercicio modifica el contador de palabras tradicional para usar los mismos criterios de reconocimiento que el ejercicio anterior:

```
%{
int numChars = 0, numWords = 0, numLines = 0;
}%

%%
if|then|else|for|begin|end      { numWords++; numChars += yyleng; }
[a-zA-Z][a-zA-Z0-9]*          { numWords++; numChars += yyleng; }
[0-9]+                          { numWords++; numChars += yyleng; }
\n                              { numLines++; numChars++; }
.                               { numChars++; }
%%

int main() {
    yylex();
    printf("%d\t%d\t%d\n", numChars, numWords, numLines);
    return 0;
}
```

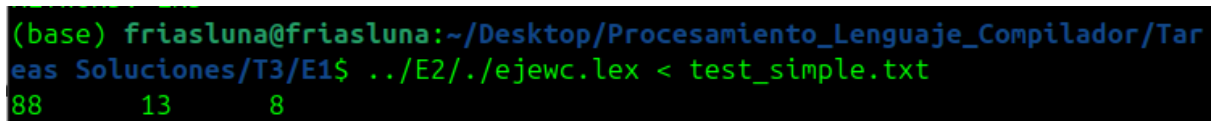
### Instrucciones de Compilación y Ejecución

```
flex ejewc.flex
gcc -o count lex.yy.c -ll
./count < test_simple.txt
```

### Salida Obtenida

```
88  13  8
```

Imagen:



```
(base) friasluna@friasluna:~/Desktop/Procesamiento_Lenguaje_Compilador/Tareas Soluciones/T3/E1$ ../E2/./ejewc.lex < test_simple.txt
88      13      8
```

### Análisis de Resultados

El programa cuenta correctamente:

- 88 caracteres totales
- 13 palabras (incluyendo palabras clave e identificadores)
- 8 líneas en el archivo

Los conteos son precisos y reflejan el contenido del archivo de entrada, considerando las palabras clave como tokens individuales.

# Ejercicio 3: Analizador Léxico Pascal

## Implementación

Este es el ejercicio más complejo, implementando un analizador léxico completo para un subconjunto de Pascal:

```
%{
#include <math.h>
%}

DIGIT      [0-9]
ID         [a-z][a-z0-9]*
STRING     \"[^\"]*\"

%%
{DIGIT}+           { printf("INTEGER: %s (%d)\n", yytext,
atoi(yytext)); }
{DIGIT}+"."{DIGIT}* { printf("FLOAT: %s (%g)\n", yytext,
atof(yytext)); }
if|then|begin|end|procedure|function|program|var|integer|real|string|wh
ile|do|repeat|until {
                    printf("KEYWORD: %s\n", yytext); }
{ID}               { printf("IDENTIFIER: %s\n", yytext); }
{STRING}           { printf("STRING: %s\n", yytext); }
":="|"+"|"-|"*"|"/"|"="|"<"|">"|"<="|">=" { printf("OPERATOR: %s\n",
yytext); }
"\"[^\"]*\n\""      /* ignore comments */
[ \t\n]+           /* ignore whitespace */
;                  { printf("SEMICOLON\n"); }
":"                { printf("COLON\n"); }
"("                { printf("OPEN_PAREN\n"); }
")"                { printf("CLOSE_PAREN\n"); }
.                  { printf("UNRECOGNIZED: %s\n", yytext); }
%%

int main(int argc, char **argv) {
    if (argc > 1) {
        if (!(yyin = fopen(argv[1], "r"))) {
            perror(argv[1]);
            return 1;
        }
    }
    return yylex();
}
```

## Archivo de Prueba (test\_pascal.txt)

```
program ejemplo;
var
    x, y: integer;
    z: real;
    mensaje: string;

procedure calcular(a: integer; b: real);
begin
    x := 42;
    y := x + 10;
    z := 3.14159;
    mensaje := "Hola mundo";

    if x > 40 then
        begin
            z := z * 2.0;
        end;

    while y > 0 do
        begin
            y := y - 1;
        end;
    end;

end;

{ Este es un comentario de prueba }
```

## Instrucciones de Compilación y Ejecución

```
flex pascal.flex
gcc -o pasc lex.yy.c -ll
./pasc < test_pascal.txt
```

## Salida Obtenida

La salida muestra una clasificación detallada de cada token encontrado en el código, incluyendo:

- Palabras clave (program, var, procedure, etc.)
- Identificadores (ejemplo, x, y, z, etc.)
- Números (enteros y flotantes)
- Operadores (:=, +, -, \*, >)
- Símbolos especiales (paréntesis, punto y coma, etc.)
- Strings

Los comentarios son correctamente ignorados.

### Imagen:

```
(base) friasluna@friasluna:~/Desktop/Procesamiento_Lenguaje_Compilador/Tareas Soluciones/
T3/E3$ ./pascal.flex < test_pascal.txt
KEYWORD: program
IDENTIFIER: ejemplo
SEMICOLON
KEYWORD: var
IDENTIFIER: x
UNRECOGNIZED: ,
IDENTIFIER: y
COLON
KEYWORD: integer
SEMICOLON
IDENTIFIER: z
COLON
KEYWORD: real
SEMICOLON
IDENTIFIER: mensaje
COLON
KEYWORD: string
SEMICOLON
KEYWORD: procedure
IDENTIFIER: calcular
OPEN_PAREN
IDENTIFIER: a
COLON
KEYWORD: integer
SEMICOLON
IDENTIFIER: b
COLON
KEYWORD: real
CLOSE_PAREN
SEMICOLON
KEYWORD: begin
IDENTIFIER: x
OPERATOR: :=
INTEGER: 42 (42)
```

```
SEMICOLON  
IDENTIFIER: z  
OPERATOR: :=  
FLOAT: 3.14159 (3.14159)  
SEMICOLON  
IDENTIFIER: mensaje  
OPERATOR: :=  
STRING: "Hola mundo"  
SEMICOLON  
KEYWORD: if  
IDENTIFIER: x  
OPERATOR: >  
INTEGER: 40 (40)  
KEYWORD: then  
KEYWORD: begin  
IDENTIFIER: z  
OPERATOR: :=  
IDENTIFIER: z  
OPERATOR: *  
FLOAT: 2.0 (2)  
SEMICOLON  
KEYWORD: end  
SEMICOLON  
KEYWORD: while  
IDENTIFIER: y  
OPERATOR: >  
INTEGER: 0 (0)  
KEYWORD: do  
KEYWORD: begin  
IDENTIFIER: y  
OPERATOR: :=  
IDENTIFIER: y  
OPERATOR: -  
INTEGER: 1 (1)  
SEMICOLON  
KEYWORD: end  
SEMICOLON  
KEYWORD: end  
SEMICOLON
```

## Análisis de Resultados

El analizador léxico demuestra una capacidad robusta para:

1. Identificar y clasificar correctamente todos los tipos de tokens en el lenguaje
2. Manejar diferentes tipos de datos (enteros, flotantes, strings)
3. Reconocer operadores y símbolos especiales
4. Ignorar apropiadamente comentarios y espacios en blanco
5. Mantener el contexto y la estructura del programa



## Conclusiones

Los tres ejercicios demuestran una progresión en complejidad y funcionalidad:

1. El primer ejercicio establece las bases para el reconocimiento de tokens básicos
2. El segundo ejercicio aplica estos conceptos en un contexto de análisis estadístico
3. El tercer ejercicio integra todos los conceptos en un analizador léxico completo

La implementación exitosa de estos ejercicios demuestra la potencia y flexibilidad de Flex como herramienta para el análisis léxico, así como su utilidad en el desarrollo de compiladores y procesadores de lenguajes.

## Recomendaciones

Para futuros desarrollos, se podrían considerar las siguientes mejoras:

1. Implementar manejo de errores más robusto
2. Agregar soporte para más características del lenguaje Pascal
3. Integrar el analizador léxico con un analizador sintáctico

# Análisis y Explicación de la Gramática del Juego de Tenis

## Introducción

El juego de tenis posee un sistema de puntuación único que puede ser representado formalmente mediante una gramática. Esta representación nos permite entender la estructura y las reglas del juego de una manera sistemática y precisa. En este reporte, analizaremos la gramática que describe el sistema de puntuación del tenis, su implementación y las características que la hacen especial.

## Estructura de la Gramática

La gramática del tenis se construye sobre una serie de estados y transiciones que reflejan la progresión natural del juego. Los componentes fundamentales son:

### Reglas de Producción Básicas

La gramática comienza con las siguientes producciones fundamentales:

```
Game -> Score Finish
Score -> Points | Advantage
Points -> Love | Fifteen | Thirty | Forty
Advantage -> Ad-in | Ad-out
Finish -> ServerWins | OpponentWins
```

Estas reglas establecen la estructura básica del juego, donde cada partida debe terminar en una victoria para alguno de los jugadores.

### Sistema de Transiciones

Las transiciones entre estados se realizan mediante dos símbolos terminales:

- 's': representa un punto ganado por el servidor
- 'o': representa un punto ganado por el oponente

Por ejemplo:

```
Love -> Fifteen      (transición con 's')
Love -> Love-Fifteen (transición con 'o')
```

### Manejo de Situaciones Especiales

La gramática incluye reglas específicas para manejar situaciones particulares del tenis:

#### Empates (Deuce)

```
Forty-All -> Deuce
Deuce -> Ad-in    (con 's')
```

Deuce -> Ad-out (con 'o')

### Sistema de Ventaja

Ad-in -> ServerWins (con 's')  
Ad-in -> Deuce (con 'o')  
Ad-out -> OpponentWins (con 'o')  
Ad-out -> Deuce (con 's')

## Características Especiales de la Gramática

### Dependencia del Contexto

La gramática del tenis exhibe características de dependencia del contexto, particularmente en el sistema de deuce. Esta dependencia se manifiesta en la necesidad de ganar dos puntos consecutivos después de un empate a 40 puntos.

### Estado y Memoria

El sistema requiere mantener un estado que recuerde:

- La puntuación actual de cada jugador
- Si el juego está en situación de deuce
- Quién tiene la ventaja (si aplica)

### Determinismo

La gramática es determinista: para cada estado y entrada (punto ganado), existe una única transición posible, lo que garantiza que el juego siempre progresa de manera predecible.

## Implementación y Verificación

Para implementar esta gramática en un sistema computacional, podríamos utilizar un autómata finito determinista (DFA) que:

1. Mantenga el estado actual del juego
2. Procese cada punto ganado ('s' u 'o')
3. Realice las transiciones apropiadas
4. Verifique las condiciones de victoria

Un ejemplo de procesamiento de una secuencia:

Estado Inicial: Love

Secuencia: s o s s s

Procesamiento:

Love -(s)-> Fifteen -(o)-> Fifteen-All -(s)-> Thirty-Fifteen -(s)->  
Forty-Fifteen -(s)-> ServerWins

# Ventajas de la Representación Gramatical

La representación mediante gramática ofrece varios beneficios:

1. Formalización clara de las reglas del juego
2. Facilidad para verificar la validez de secuencias de puntuación
3. Base para implementar sistemas de puntuación automatizados
4. Herramienta para enseñar y entender el sistema de puntuación

## Conclusiones

La gramática del tenis representa un ejemplo interesante de cómo un sistema de reglas aparentemente complejo puede ser formalizado de manera precisa y sistemática. Sus características especiales, como el manejo de deuce y el sistema de ventaja, la convierten en un caso de estudio valioso para entender gramáticas dependientes del contexto y sistemas de estados finitos.

Esta formalización no solo tiene valor académico, sino que también proporciona una base sólida para:

- Implementar sistemas de puntuación automáticos
- Verificar la correctitud de secuencias de juego
- Enseñar las reglas del tenis de manera sistemática
- Analizar patrones y estrategias de juego

La gramática presentada captura efectivamente todas las reglas y sutilezas del sistema de puntuación del tenis, proporcionando una herramienta valiosa para su comprensión y análisis formal.