Prácticas MC 3°B

Alejandro Anglada Álvarez

El documento tiene muchas páginas porque adjunto muchas capturas, no porque sea extenso. Enlace al repositorio aquí.

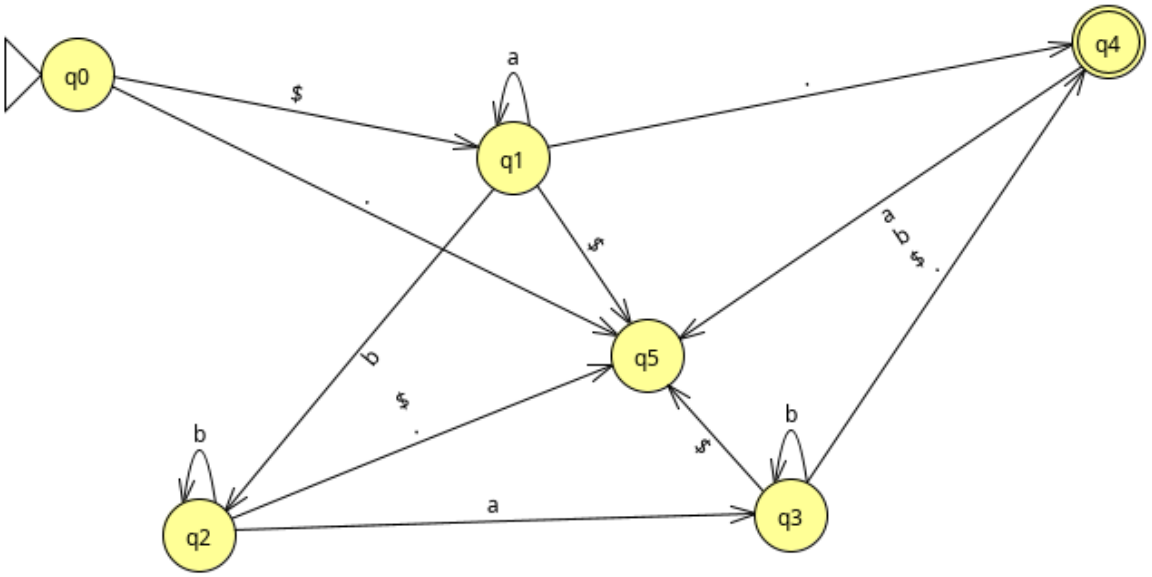
1.

Obtener un autómata finito determinístico minimal, la gramática regular y la expresión regular para implementar la parte del receptor de un protocolo de comunicación que solamente tiene estructura léxica.

Para empezar, debemos definir cómo será la estructura del receptor de dicho protocolo. En mi caso, he decidido que los datos que recibe el receptor son cadenas de un dólar ($), *a*'s, *b*'s y un punto (.) que cumplen que:

1. Empiezan con un símbolo de dólar ($) para que el receptor entienda que lo que le ha llegado es una cadena que debe ser procesada. Todo dato que llegue antes que el símbolo de dólar no será leído; así el receptor es menos susceptible a ataques de desbordamiento de *buffer* (siempre y cuando el atacante no conozca el símbolo de comienzo, claro).
2. Acaben con un punto (.) para que el receptor sepa que ahí es donde acaba el mensaje. Si luego de leer un punto la cadena sigue, se debe de rechazar, pues los datos han sido corrompidos (bien por una interferencia, o por un ataque *man-in-the-middle* en el que el hombre de en medio ha intentado inyectar código malicioso, o vete a saber qué, esto es para complicarlo un poco).
3. En el momento en el que se reciba una *b*, el receptor debe contar el número de *a*'s que recibe. Si es impar, el emisario es el que debe ser y todo puede continuar. Si no, es posible que el emisario haya sido suplantado y debe rechazar lo que haya recibido. Cabe decir que un mensaje puede ser puramente *a*'s, luego el primer estado (antes de pasar a contar paridad por haber encontrado una *b*) es final. Además, como el número de *a*'s después de la primera *b* debe ser impar, el estado de después de leer una *b* no puede ser final (hay cero *a*'s después de *b*, y 0 mód 2 = 0 (par))

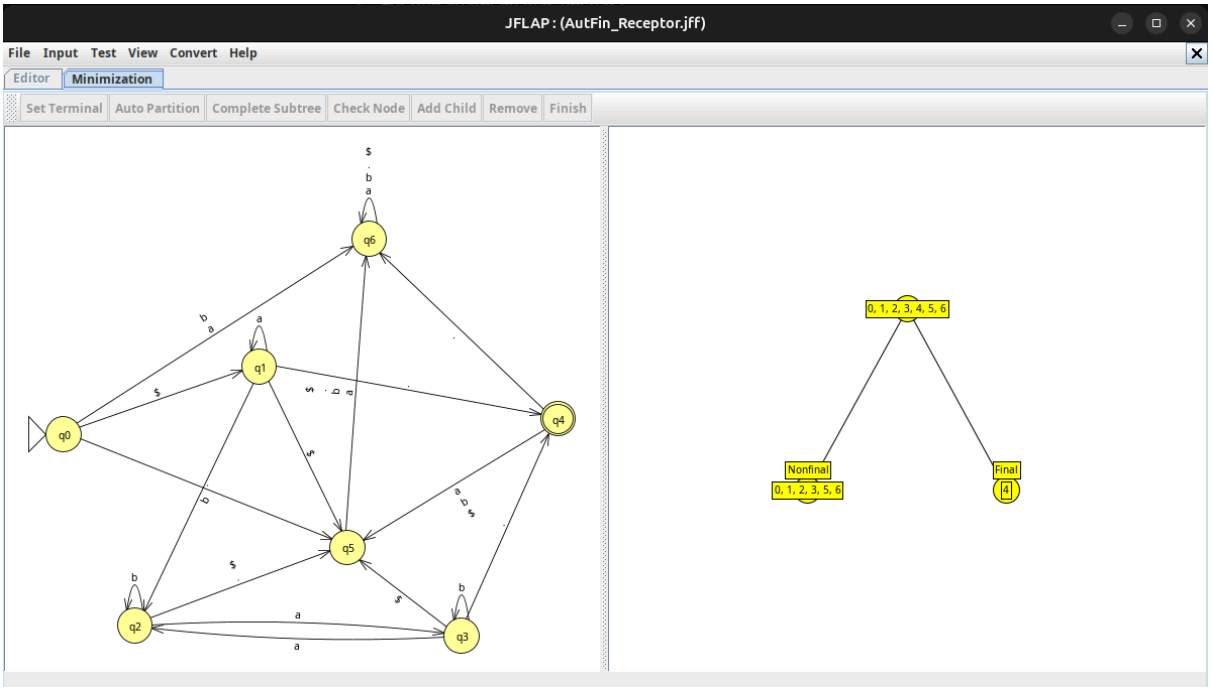
Ahora que las normas ya están propuestas, pasemos a la implementación. Primero, el autómata.



Esta es mi primera implementación. Al tratar de convertirla a determinista, JFLAP me ha dicho que ya lo es, luego ya cumplo que sea un autómata determinístico finito. Funciona así:

- **q5 es un estado "basura"** al que se llega siempre que se haya encontrado una incongruencia (no hay un número impar de *a*'s, la comunicación se ha acabado antes de empezar, después de haber acabado la comunicación se ha leído algo más o en un nodo de procesamiento común se ha encontrado un símbolo de comienzo de lectura ($)).
- **q4 es el nodo final** donde se debe llegar si todo acaba bien. Sólo se puede llegar a él si se ha acabado la comunicación antes de encontrar una *b* o si el número de *a*'s es impar, y entonces se ha acabado la comunicación.
- **q0 es el nodo de inicio**. Sólo se puede ir desde él o bien a q1 (véase ahora) o a q5 (a este último si se lee un . antes de iniciar la lectura).
- **q1, q2 y q3 son nodos de lectura**. q1 lee *a*'s hasta que encuentre o un carácter que no debe ($, pasaría a q5), o un punto (pasaría a q4) o una *b* (pasaría a q2). q2 lee *b*'s hasta que encuentre un dólar (se iría a q5) o hasta que encuentre una *a* (pasaría a q3). a3 lee *b*'s hasta que encuentre un dólar (pasaría a q5), un punto (pasaría a q4) o una *a* (pasaría a q2).

Al decirle a JFLAP que lo convierta a su forma minimal, ha resultado en esto (después de manipularlo para que los nodos se asemejen lo máximo posible a los originales):



Y creo que sé por qué tiene un nodo más. Al parecer, JFLAP trata de evitar estados "basura" que no tienen transiciones que vayan de dicho estado a cualquier otro, luego añade otro estado "basura" que ya sí que es cíclico. Eso significa que mi primera implementación ya era minimal, en cierto modo.

Al convertir la forma minimal a una gramática, nos queda esta serie de producciones:

LHS		RHS
S	→	.E
A	→	aA
B	→	.E
C	→	bC
B	→	bB
D	→	λ
D	→	bE
B	→	$E
D	→	aE
D	→	$E
C	→	aB
B	→	aC
S	→	$A
A	→	bB
C	→	.D
C	→	$E
A	→	.D
A	→	$E

Si bien son muchas producciones, es cierto que se trata de una **gramática regular** cuyos símbolos no terminales son **{S**, A, B, C, D**}**; y sus símbolos terminales son {$, a, b, .}. Nótese que la E es un "valor sumidero" que añade JFLAP para representar mi estado "basura". Toda cadena que para desarrollarse tenga que pasar por E, no podrá desarrollarse. Surte el mismo efecto eliminar todas las producciones que contengan la E. Aquí estaría la gramática sin estas producciones basura (razonablemente más pequeña):

LHS		RHS
S	→	$A
A	→	aA
C	→	bC
B	→	bB
D	→	λ
C	→	aB
B	→	aC
A	→	bB
C	→	.D
A	→	.D

Lo único que resta es encontrar una expresión regular que pueda generar cadenas aceptadas por esta gramática. No es una tarea demasiado difícil, pues solo hay que asegurar que la lectura de *a*'s luego de la primera *b* sea impar. Nos resulta la siguiente expresión regular:

$(a* + (a*b(b*ab*a)*b*ab*)).

A pesar de que parezca poco intuitivo, tiene mucho sentido. Primero, se empieza con un sólo dólar y se acaba con un sólo punto. Hasta ahí, no hay mucho problema. Luego, la parte de dentro del primer paréntesis, es también muy sencilla: o bien se leen sólo *a*'s, lo que es perfectamente válido, o bien se lee un número indeterminado (pero finito) de *a*'s, y luego una *b*, y aquí viene la parte más "abstracta": después de esto, se deben leer un número de cadenas que posean una cantidad par de *a*'s y una cantidad finita de *b*'s, seguido de una cantidad finita de *b*'s y una última *a* (esta última *a* siempre debe aparecer para asegurar la imparidad de la cadena).

2.
Montar un escáner para reconocer cadenas mediante el uso de un analizador léxico.

Para esta práctica he decidido implementar (usando *ylex*) un clasificador de idiomas basado en tendencias léxicas. Existen unos conceptos en los idiomas llamados *n-gramas* y *frecuencias* que miden con qué frecuencia aparecen ciertas agrupaciones de *n* letras en ciertos idiomas. Entonces, conociendo algunos *n-gramas* frecuentes de una serie de idiomas, es posible detectar con cierto porcentaje de fiabilidad de qué idioma se trata el texto que está siendo analizando léxicamente.

Para empezar, en el repositorio, dentro de la carpeta "Pruebas" hay 4 documentos de texto plano, cada uno probando con diferentes libros influyentes de cada idioma (que, salvo el de francés e italiano, he tenido la oportunidad de leerme en español y los he disfrutado enormemente). Son "El Ingenioso Hidalgo Don Quixote de la Mancha", de Miguel de Cervantes; "Zadig, o el destino", de Voltaire; "La Vida Nueva", de Dante; y "Así habló Zaratustra", de Friedrich Nietzsche.

Es cierto que, con esos textos, no ha habido problema en detectar sus idiomas salvo con el de italiano, que en una primera versión lo confundía con alemán porque fui muy laxo con las reglas del alemán, y en la versión final lo confunde (por muy poco) con el español, con lo que me doy por satisfecho, porque al final son idiomas muy similares.

Estoy muy orgulloso del resultado. Pasaremos a revisar el código y las expresiones regulares usadas para cada idioma (recomiendo verlo mejor desde el repositorio):

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int es = 0;
int fr = 0;
int it = 0;
int de = 0;
int total_palabras = 0;
%}

LETRA    [A-Za-zÁÉÍÓÚáéíóúÑñÜü]
PALABRA  {LETRA}+
ESPACIO  [ \t\n\r]

%%

ñ[a-zA-Z]*|Ñ[a-zA-Z]*       { es += 3; total_palabras++; }
ll[a-zA-Z]*|rr[a-zA-Z]*     { es += 2; total_palabras++; }
ci[óÓ]n[a-zA-Z]*            { es += 2; total_palabras++; }  /* ción */
ando|ado|mente              { es += 1; total_palabras++; }
```

```
ç[a-zA-Z]*|â[a-zA-Z]*|ê[a-zA-Z]*|ô[a-zA-Z]*|û[a-zA-Z]* { fr += 3; total_palabras++;
}


zz[a-zA-Z]*|nn[a-zA-Z]*|tt[a-zA-Z]*|gli[a-zA-Z]* { it += 2; total_palabras++; }


sch[a-zA-Z]{2,}|tz[a-zA-Z]{2,}|ch[a-zA-Z]{2,}|pf[a-zA-Z]*{de+=2;total_palabras++; }


el[ -]?la|las? | los         { es += 1; total_palabras++; }   /* el, la, las, los
juntos */
un[ae]?|una[es]?            { es += 1; total_palabras++; }  /* un, una, unas, unas
*/
le|la|les|des|du           { fr += 1; total_palabras++; }
il|lo|gli|le|la|li|i|gli   { it += 1; total_palabras++; }
der|die|das|ein[e]?[mn]?   { de += 1; total_palabras++; }


[a-zA-ZÁÉÍÓÚáéíóúÑñÜü]+   { total_palabras++; }
[ \t\n\r]+                 { /* ignorar */ }
.                          { /* ignorar */ }
%%


int yywrap() {
    return 1;
}

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

    printf("Analizando texto...\n");
    yylex();

    printf("\n=== RESULTADOS DEL ANÁLISIS ===\n");
    printf("Total palabras procesadas: %d\n", total_palabras);
    printf("Español:  %d puntos\n", es);
    printf("Francés:  %d puntos\n", fr);
    printf("Italiano: %d puntos\n", it);
    printf("Alemán:   %d puntos\n", de);
    printf("\nPREDICCIÓN FINAL: ");

    int max = es;
```

```
    char *idioma = "ESPAÑOL";

    if (fr > max) { max = fr; idioma = "FRANCÉS"; }
    if (it > max) { max = it; idioma = "ITALIANO"; }
    if (de > max) { max = de; idioma = "ALEMÁN"; }

    if (max == 0) {
        printf("No se encontraron patrones reconocibles\n");
    } else {
        printf("%s (con %d puntos)\n", idioma, max);
    }

    printf("Total patrones encontrados: %d\n", es+fr+it+de);

    if(argc>1)fclose(yyin);
    return 0;
}
```

En la primera sección definimos las variables que vamos a usar para almacenar los datos. Son *es*, para los *tokens* españoles que detectemos; *fr*, para los franceses; *de* para los alemanes e *it* para los italianos. Además, por razones de métricas, contaremos también las palabras que hemos analizado (así es más significativo descubrir un idioma u otro; no tendría sentido que de 70.000 palabras, sólo se detecten 150 *tokens*, por ejemplo).

Luego definimos expresiones regulares para palabras, letras y espacios, que nos serán útiles para métricas y encontrar patrones de idiomas.

Analizamos las expresiones regulares del español:
La primera expresión regular se encarga de encontrar las letras *ñ* que puedan haber en el texto. Como son letras únicas del español, tienen peso triple. La segunda expresión regular busca "*ll*" o "*rr*", de nuevo apariciones propias del español. La tercera expresión regular busca la terminación "*ción*", que es muy común en sustantivos deverbales,

La expresión regular del francés busca dentro de las palabras bien cés cedillas (ç) o vocales con acentos circunflejos. Son apariciones típicas del francés.

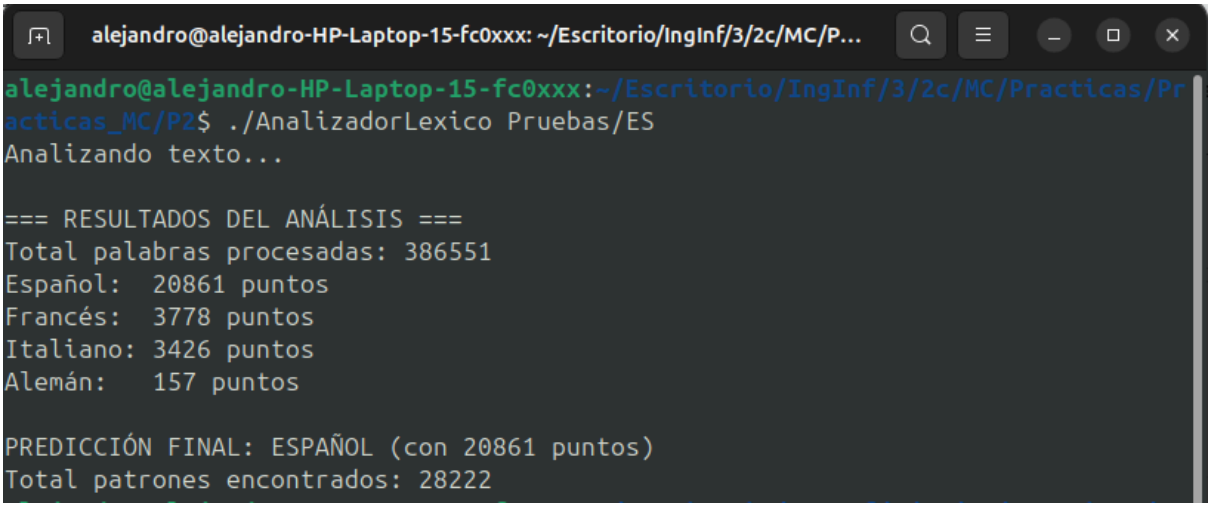
La expresión regular del italiano busca duplicidad de consonantes "*zz*", "*tt*" y el trigrama "*gli*".

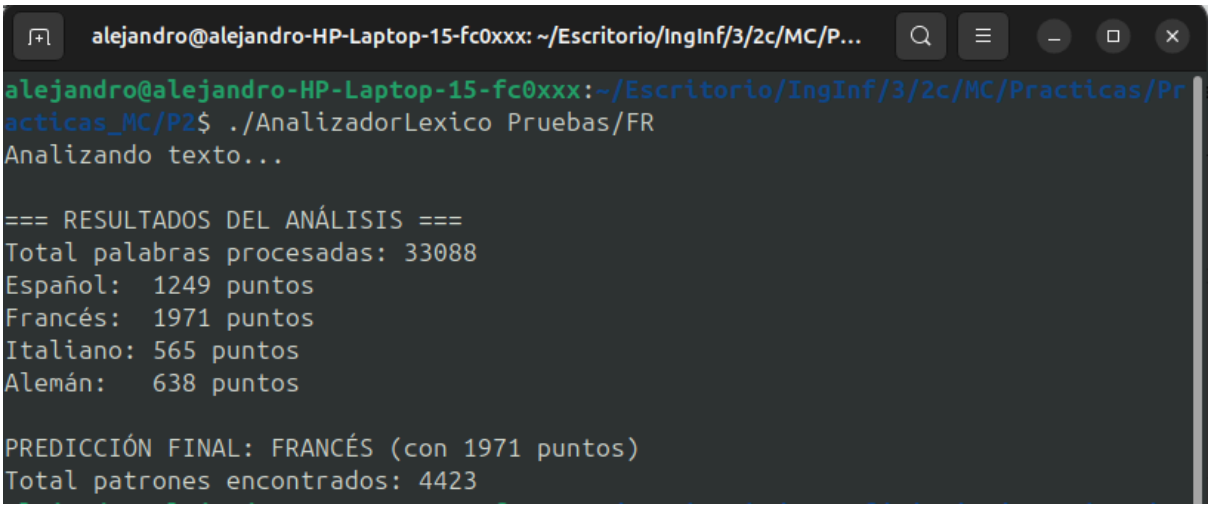
La expresión regular del alemán busca palabras que contengan sus bigramas "*tz*", "*ch*", "*pf*" o el trigrama "*sch*".

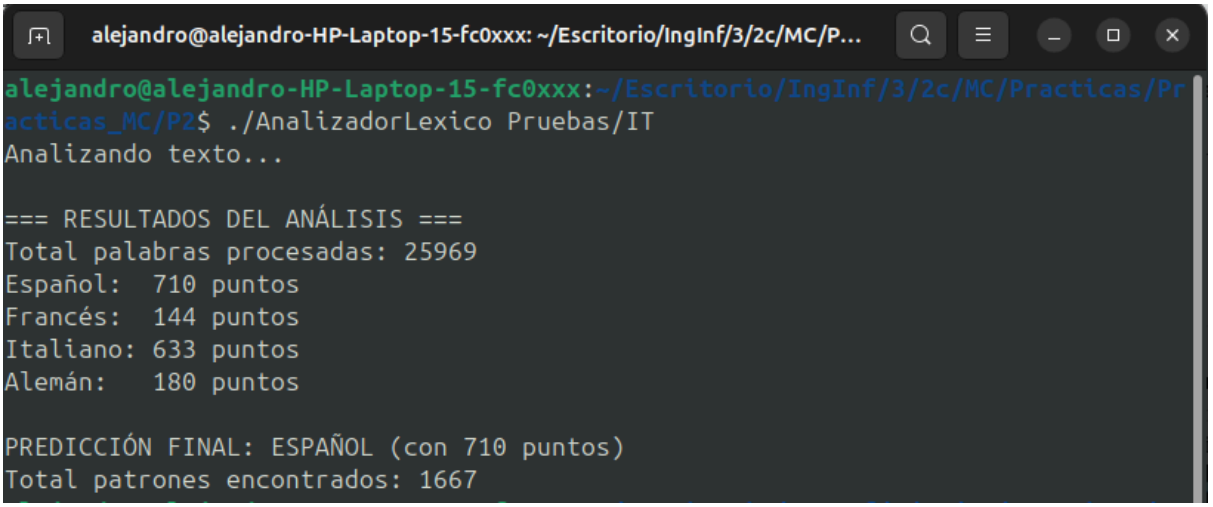
Luego, hay una sección que se encarga de buscar coincidencias exactas de palabras comunes de los idiomas, como determinantes y preposiciones. No voy a profundizar mucho, porque sus expresiones regulares son extremadamente sencillas (del tipo *término* + *término*).

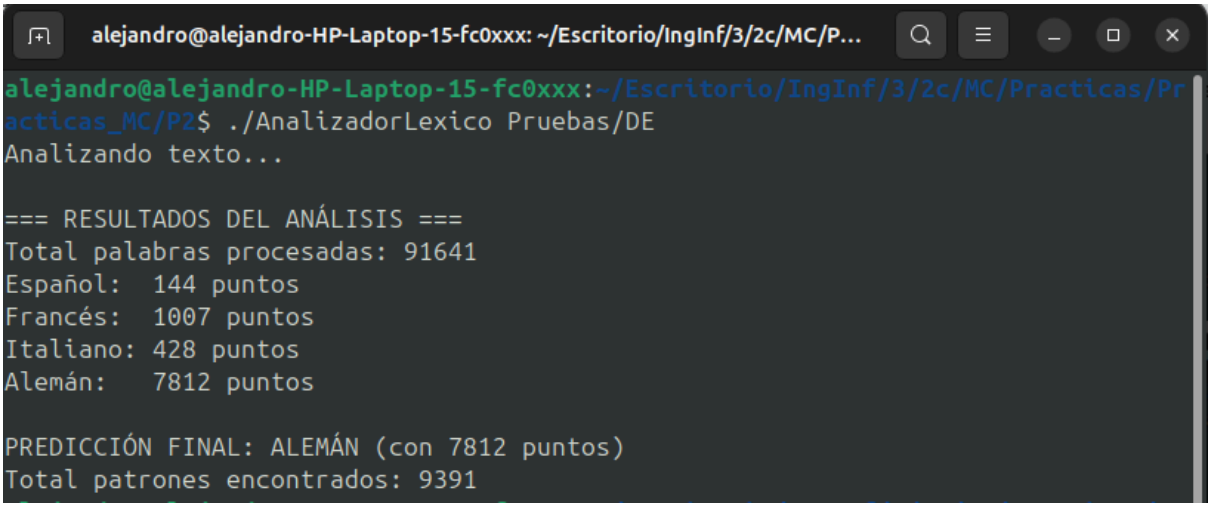
La última parte del segundo sector del archivo se encarga de consumir caracteres y palabras de escaso interés, y de contarlas de ser consumidas.

Luego, viene la lógica del código. Es simple; muestra las palabras procesadas, y luego hace comparaciones sencillas para ver de qué idiomas se han encontrado más *tokens*. Adjunto capturas de los ejemplos de ejecución con los diferentes textos de prueba:







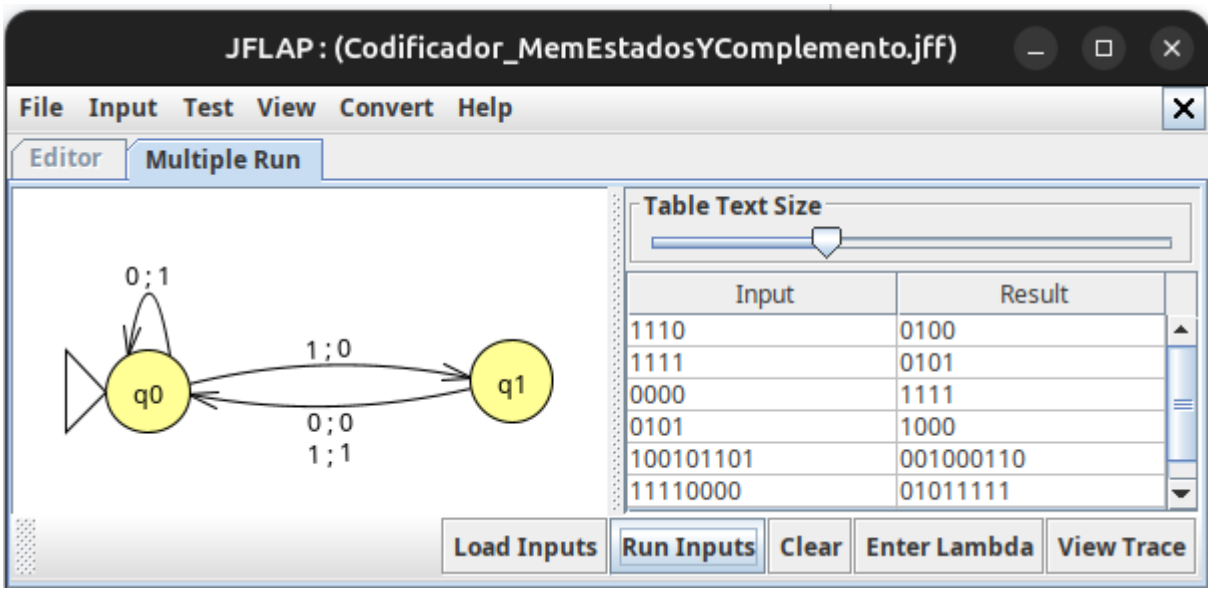


Como último apunte, es posible que la precisión del analizador léxico aumente muy considerablemente si redujese el número de *tokens* que pueden leerse del español.

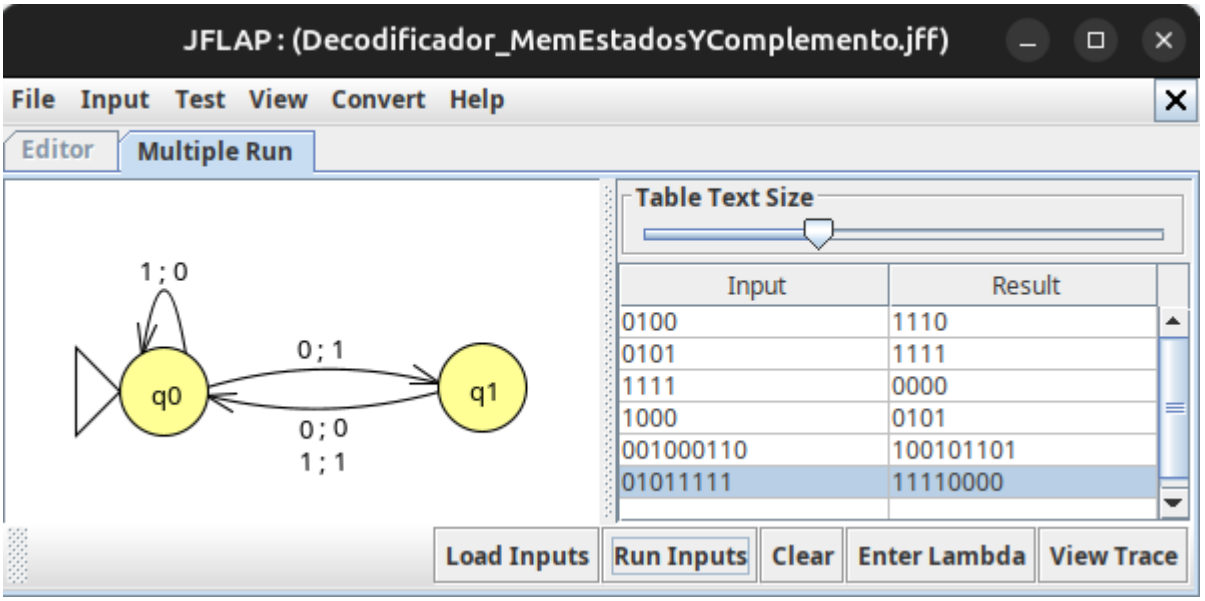
3.
Implementar una máquina de estados finitos para codificar y descodificar.

Por último, he decidido implementar una máquina de estados finita que se encargue de cifrar usando memorización, y luego le haga un complemento. La idea es que si la máquina lee un 0, lo deje tal y como está, y si encuentra un 1 cambie en función de los estados anteriores que encuentre; si sólo ha leído 0's lo escribe tal y como está, y si ha leído algún 1 lo invierte. Por último, esta cifra **siempre** será invertida, para hacer una encriptación más robusta (a dos capas) en la que es más difícil establecer una relación directa entre la cadena original y la resultante. Este último paso es tan sencillo como implementar la primera "aproximación" y luego invertir las salidas.

Se implementará con una **máquina de Mealy** porque funcionan mucho mejor y simplifican mucho el diseño para las operaciones sobre *bits*. Así es como resulta la máquina de codificación:



Ahora, implementaremos el decodificador y le pasaremos los resultados para ver que efectivamente se devuelve la cadena original. De ser así, la máquina de estados funcionaría a la perfección:



Se ve que se devuelven siempre las cadenas originales. Entonces, se puede decir que este codificador ha sido implementado con éxito. ¡Bien!