UNIVERSIDAD
PANAMERICANA®

Teoría de Lenguajes y Programación

Ejercicios Flex

Jorge Barba Pérez 0237328

Daniel Alejandro Osornio López 0244685

Índice

1 Correr los programas

Los ejercicios fueron resueltos usando Linux, para ejecutar los programas ofrecemos 2 alternativas, ambas requieren `flex` y `gcc` disponibles en el `PATH`, asi como `rm`, `cp` y `mkdir`, `echo`. A su vez se usan separadores de ruta del estilo Unix, es decir `/`.

1.1 Usando `make`

La manera más sencilla para ejecutar cualquiera de los ejercicios, si no desea pasar más argumentos que no sean los establecidos por defecto para cada ejercicio en el archivo `Makefile`, entonces ejecute en el mismo directorio del archivo `Makefile`:

```
make ejercicio_N
```

Donde en lugar de `N` pone un número de ejercicio, del `1` al `6`. Por ejemplo, para correr el ejercicio `5`:

```
make ejercicio_5
```

También puede ejecutar todos los ejercicios a la vez, con los argumentos puestos por defecto para todos, usando:

```
make todos
```

1.1.1 Ejecutando con argumentos específicos con `make`

Si desea ejecutar cualquiera de los ejercicios poniendo argumentos distintos a los establecidos por defecto usando `make` puede ejecutar:

```
make run num=N args="ARG_1 ARG_2 ..."
```

Por ejemplo, para ejecutar el ejercicio 2, sumando `100` a todos los múltiplos de `2`, leyendo del archivo `'./entradas/ejercicio1.txt'` y poniendo el resultado en el archivo `'./salidas/ejercicio2.txt'` puede usar:

```
make run num=2 args="100 2 ./entradas/ejercicio1.txt ./salidas/ejercicio2.txt"
```

Puede guiarse para saber qué argumentos pasar inspeccionando los contenidos del archivo `Makefile`.

1.1.2 Compilando con make, ejecutando manualmente

Puede saltarse la ejecución usando el *target* `build`, de la forma:

```
make build num=N
```

Y después ejecutar manualmente, todos los ejecutables se colocan en `./bin/` bajo el nombre `ejercicioN`. Por ejemplo, para compilar y correr el ejercicio 6:

```
make build num=6
./bin/ejercicio6 ./entradas/ejercicio6.txt
```

1.2 Compilación y ejecución manual

Primero solo genere el archivo `lex.yy.c` usando flex, por ejemplo, para el ejercicio 1:

```
flex -L src/ejercicio1.lex
```

Después solo compile con `gcc`:

```
gcc lex.yy.c
```

Por último ejecute el objeto generado:

```
./a.out ./entradas/ejercicio1.txt ./salidas/ejercicio1.txt
```

2 Ejercicios

Para cada ejercicio incluimos una breve descripción de la forma en que se resolvió. También incluimos el código para la resolución con comentarios para explicar a profundidad los elementos relevantes del código.

Nota: El código en este documento incluye acentos y distintos caracteres que pueden causar problemas si se intenta copiar y pegar en Visual Studio o algún editor de texto. Para copiar/ejecutar el código le recomendamos ver los archivos de código fuente disponibles en el zip adjunto, están libres de este tipo de caracteres.

2.1 Ejercicio 1

Elaborar un programa en flex que copie el archivo de entrada en uno de salida, sumando 5 a todo número positivo que sea múltiplo de 4.

```
make ejercicio_1
```

Para resolverlo escaneamos para encontrar todos los números que sean múltiples de 2 (p. ej. terminan en `0`, `2`, `4`, etc.).

Para todos los números identificados, verificamos que sean múltiples de 4, le sumamos 5 y contamos el número de digitos del resultado para asegurarnos que nuestros buffer `nuevo[]` puede contener el N número de caracteres.

El buffer `nuevo[]` almacena la cadena de caracteres del resultado de $N + 5$ para poder escribirlo en el archivo de salida. Para lograr esto remplazamos los contenidos de `yytext` y `yyleng`, después realizamos un `ECHO` normal.

Hacemos esto pues `yytext` apunta al texto actual (mismo que se muestra en la salida al hacer `ECHO`) y `yyleng` contiene el tamaño de `yytext`.

```
%{
  #include <stdio.h>
  #pragma GCC diagnostic ignored "-Wunknown-pragmas"
  #pragma warning(disable:4996 6011 6385 4013)
  #pragma GCC diagnostic ignored "-Wunused-function"
  #pragma GCC diagnostic ignored "-Wsign-compare"

  /*
    Para resolver el problema establecemos un arreglo de hasta 25 dígitos para
    contener el texto del resultante de sumar N + 5.

    Por ejemplo 8 + 5 es 13, que necesita 2 caracteres para ser almacenado,
    para posteriormente ser escrito en el archivo de salida

    Incluimos un bytes más para tener para el delimitador de fin de cadena '\0'
  */
  #define MAX_DIGITOS 25
  char nuevo[MAX_DIGITOS+1] = {0};
%}

%option noyywrap

%%
[48]|[1-9]+[24680] {
  /*
    Solo nos molestamos en verificar, en el regex, números que terminen en múltiplos
    de 2 o que sean 4 u 8. Hacemos esto pues sabemos que cualquier otro número no
    puede ser multiplo de 4, ya que no es múltiplo de 2.
  */
  unsigned int numero = atoi(yytext);

  if (numero%4 == 0) {
    numero+=5;

    unsigned int digitos = 0;
    unsigned int tnum = numero;

    /*
      Contamos el número de dígitos requeridos para almacenar la cadena del resultado.
    */
    do {
      tnum/=10;
      digitos++;
    } while (tnum > 0);

    /*
      Si el número de digitos sobrepasa el buffer de `MAX_DIGITOS` que tenemos,
      para evitar un overflow, no remplazamos el texto en el archivo de salida.
    */
    if (digitos > MAX_DIGITOS) {
       printf("Error: %s + 5 (%d) tiene mas de %d digitos (%d), saltando.\n", yytext,
numero, MAX_DIGITOS,  digitos);
    } else {
      sprintf(nuevo, "%d", numero);
```

```
      /*
        Remplazamos `yytex` y `yyleng`, que contenía el número original del archivo
        de entrada, por el resultado de N + 5, de forma que cuando se ejecute `ECHO`
        se escriba en el archivo de salida el valor como cadena de caracteres de N + 5.
      */
      yytext = nuevo;
      yyleng = digitos;
    }
  }

  ECHO;
}
.|\n ECHO;
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  /*
    Esperamos exactamente 2 argumentos, el archivo de entrada de donde se leerán
    los números múltiplos de 4, y el de salida donde se remplazarán por
    el resultado de N + 5.
  */
  if (argc != 2) {
     puts("Debe especificarse un archivo de entrada y otro de salida.\nEjemplo: ./
ejercicio1 entrada.txt salida.txt");
    exit(1);
  }

  FILE * in = fopen(argv[0], "r");
  if (in == NULL) {
    printf("Fallo al abrirse el archivo '%s'.\n", argv[0]);
    exit(1);
  }

  FILE * out = fopen(argv[1], "w");
  if (out == NULL) {
    printf("Fallo al crear/abrir el archivo '%s'.\n", argv[1]);
    exit(1);
  }
  yyin = in;
  yyout = out;

  printf("Leyendo de: %s\n", argv[0]);
  printf("Salidas en: %s\n", argv[1]);

  yylex();
}
```

2.2 Ejercicio 2

Elabora un programa en flex que copie el archivo de entrada en uno de salida, sumando N1 a todo número positivo que sea múltiplo de N2, donde N1 y N2 son dos números pasados como argumentos desde la línea de comandos.

```
make ejercicio_2
```

Este segundo ejercicio es muy similar al primero, solo que ahora verificamos que los números enteros encontrados sean múltiplos de `N2`, sumando `N1` a todos los casos que se cumplan.

Lo único nuevo en el código es la adición de las variables globales `unsigned int num1;` y `unsigned int num2;` para almacenar ambos números pasados como argumento desde la linea de comandos.

```
%{
  #include <stdio.h>
  #pragma GCC diagnostic ignored "-Wunknown-pragmas"
  #pragma warning(disable:4996 6011 6385 4013)
  #pragma GCC diagnostic ignored "-Wunused-function"
  #pragma GCC diagnostic ignored "-Wsign-compare"

  /*
    Usamos la misma estrategia que del ejercicio 1, donde colocamos la representación
    como texto de $N + N1$ para todos los números que sean múltiplos de $N2$ para
    después escribirlo al archivo de salida aprovechando `ECHO`
  */
  #define MAX_DIGITOS 25
  char nuevo[MAX_DIGITOS+1] = {0};

  /*
    Variables para contener N1 y N2
  */
  unsigned int num1;
  unsigned int num2;
%}

%option noyywrap

%%
[1-9][0-9]* {
  /*
    En este caso si nos interesa buscar cualquier número entero.
  */
  unsigned int numero = atoi(yytext);

  if (numero%num2 == 0) {
    unsigned numero+=num1;

    unsigned int digitos = 0;
    unsigned int tnum = numero;
    do {
      tnum/=10;
      digitos++;
    } while (tnum > 0);

    /*
      Verificamos que no se den buffer overflows
    */
    if (digitos > MAX_DIGITOS) {
       printf("Error: %s + 5 (%d) tiene mas de %d digitos (%d), saltando.\n", yytext,
numero, MAX_DIGITOS,  digitos);
    } else {
```

```
      sprintf(nuevo, "%d", numero);

      yytext = nuevo;
      yyleng = digitos;
    }
  }

  ECHO;
}
.|\n ECHO;
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  if (argc != 4) {
    puts("Debe especificarse N1, N2, el archivo de entrada y el de salida.\nEjemplo: ./
ejercicio2 100 2 entrada.txt salida.txt");
    exit(1);
  }

  num1 = atoi(argv[0]);
  num2 = atoi(argv[1]);

  if (num2 == 0) {
    printf("Error: N2 no puede ser 0\n");
    exit(1);
  }

  FILE * in = fopen(argv[2], "r");
  if (in == NULL) {
    printf("Fallo al abrirse el archivo '%s'.\n", argv[2]);
    exit(1);
  }

  FILE * out = fopen(argv[3], "w");
  if (out == NULL) {
    printf("Fallo al crear/abrir el archivo '%s'.\n", argv[3]);
    exit(1);
  }
  yyin = in;
  yyout = out;

  printf("Sumando %d a los multiplos de %d\n", num1, num2);
  printf("Leyendo de: %s\n", argv[2]);
  printf("Salidas en: %s\n", argv[3]);

  yylex();
}
```

2.3 Ejercicio 3

Elaborar un programa de flex que reciba un archivo de texto y cuente el número de caracteres, palabras y líneas que contiene.

```
make ejercicio_3
```

Para resolver este problema tenemos que buscar y realizar para:

- **Palabra**: Consideramos como palabra cualquier cadena de caracteres que contiene números y letras del abecedario, sea minúscula o mayúscula.

  Cuando encontramos una palabra aumentamos +1 el conteo de palabras y aumentamos +`yyleng` el conteo de caracteres.

  Usamos `yyleng` pues es la variable que expone `flex` para informar el tamaño de la cadena que ha sido capturada en el *scan* usando la expresión regular que definimos.
- **Salto de linea**: Es un carácter (+1) y es un salto de linea, o sea, que al número de lineas del archivo +1.
- **Cualquier otro caracter**: Aumenta +1 al conteo de caracteres.

```
%{
    #include <stdio.h>
    #pragma GCC diagnostic ignored "-Wunknown-pragmas"
    #pragma warning(disable:4996 6011 6385 4013)
    #pragma GCC diagnostic ignored "-Wunused-function"
    #pragma GCC diagnostic ignored "-Wsign-compare"

    /*
      Los contadores de apariciones.
      Usamos size_t para poder almacenar el máximo valor posible.
    */
    size_t caracteres = 0;
    size_t lineas = 0;
    size_t palabras = 0;
%}

%option noyywrap
%option caseless

%%
\n lineas++; caracteres++;
. caracteres++;
[a-z0-9]+ {
    /*
       Usamos `%option caseless` para poder especificar que queremos que `EXPR` haga
_match_
      sin importar si se trata de mayusculas o minusculas con la bandera `i`.
    */
    caracteres += yyleng;
    palabras++;
}
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  if (argc != 1) {
    puts("Debe especificarse unicamente un archivo de entrada.\nEjemplo ./ejercicio3
entrada.txt");
    exit(1);
```

```
  }


  FILE * in = fopen(argv[0], "r");
  if (in == NULL) {
    printf("Fallo al abrirse el archivo '%s'.\n", argv[0]);
    exit(1);
  }

  yyin = in;

  /*
    Al correr el escaner se popularán los contadores almacenados
    en variables globales. Al terminar los podemos imprimir.
  */
  yylex();

  printf("  Contenidos: %s\n", argv[0]);
  printf("  Caracteres: %zu\n", caracteres);
  printf("      Lineas: %zu\n", lineas);
  printf("    Palabras: %zu\n", palabras);
}
```

2.4 Ejercicio 4

Elaborar un programa flex que reciba un archivo de texto y una palabra y cuente el número veces que aparece dicha palabra en el archivo.

```
make ejercicio_4
```

Consideramos como palabra toda cadena de caracteres que tiene solo letras del abecedario, sin tomar en cuenta si son mayúsculas o minúsculas.

Para lograr esto usamos una opción para el patrón `[a-z]+` donde indicamos que no queremos ser *case-sensitive*, es decir, no deseamos darle importancia a si es mayúscula o minúscula [1], lo hacemos unsando `%option caseless`, que indica *case-insensitive* y `EXPR` es `[a-z]+`.

```
%{
  #include <stdio.h>
  #pragma GCC diagnostic ignored "-Wunknown-pragmas"
  #pragma warning(disable:4996 6011 6385 4013)
  #pragma GCC diagnostic ignored "-Wunused-function"
  #pragma GCC diagnostic ignored "-Wsign-compare"

  /*
   En `palabra` almacenamos el apuntador a la palabra a buscar en el archivo de entrada.
   El contador de apariciones usa `size_t` para almacenar el maximo valor posible.
  */
  char * palabra = NULL;
  size_t apariciones = 0;
%}

%option noyywrap
%option caseless

%%
[a-z]+ {
```

```
  /*
    Si encontramos una palabra, entonces verificamos si es igual a la palabra que
    estamos buscando. Recordemos que strcmp compara dos cadenas terminadas con
    un byte nulo. Si devuelve 0 entonces son idénticas, por lo tanto,
    encontramos una aparicón de la palabra objetivo.

    Para hacer _match_ de palabras con mayusculas y minusculas usamos la bandera `i`
    en (?BANDERAS:EXPR).
  */
  if (strcmp(yytext, palabra) == 0) ++apariciones;
}
.|\n {}
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  if (argc != 2) {
   puts("Debe especificarse una palabra y un archivo de entrada.\nEjemplo: ./ejercicio4
perro entrada.txt");
    exit(1);
  }


  FILE * in = fopen(argv[1], "r");
  if (in == NULL) {
    printf("Fallo al abrirse el archivo '%s'.\n", argv[1]);
    exit(1);
  }

  palabra = argv[0];
  yyin = in;

  yylex();

  printf("La palabra '%s' aparece %zu veces en el archivo de entrada %s\n", palabra,
apariciones, argv[1]);
}
```

2.5 Ejercicio 5

Elabora un analizador que reemplace una palabra por otra en un archivo de entrada. Ambas palabras, así como el nombre del archivo deberán ser introducidos por el usuario, bien a través de la línea de comandos o cuando el usuario ejecute el programa.

```
make ejercicio_5
```

Para poder remplazar todas las ocurrencias de una palabra por otra en un archivo llegamos a la conclusión de que necesitamos un archivo temporal donde podamos poner el resultado.

De otra forma, si intentamos abrir el archivo con lectura y escritura `r+` y asignarlo a `yyin` y `yyout` el resultado es incorrecto. Solo pensar en leer/escribir al mismo tiempo *se siente* mal.

Es entonces, que nuestra solución crea un archivo temporal `.tempejer5` en modo escritura `w` para borrar en caso de que ya exista (p. ej. porque se interrumpió el programa).

Se asigna a `yyout` el *handler* del archivo `.tempejer5`, y al terminar el proceso se renombra como el archivo original, efectivamente remplazandolo.

El resultado es que tenemos un archivo con la misma ruta que el archivo de entrada original, pero con todas las coincidencias de la palabra deseada remplazadas.

```
%{
  #include <stdio.h>
  #pragma GCC diagnostic ignored "-Wunknown-pragmas"
  #pragma warning(disable:4996 6011 6385 4013)
  #pragma GCC diagnostic ignored "-Wunused-function"
  #pragma GCC diagnostic ignored "-Wsign-compare"

  char * palabra = NULL;
  char * remplazo = NULL;
  size_t tamano = 0;
%}

%option noyywrap
%option caseless

%%
[a-z]+ {
  /*
    Nuevamente usamos la opción `i` para ignorar si se trata de una letra
    mayúscula o minúscula y hacer _match_ de cualquier caso.
  */
  if (strcmp(yytext, palabra) == 0) {
    /*
      Recordemos que no vale con solo cambiar `yytext`. Si el valor en `yyleng`
      es incorrecto, solo se copiará el número de caracteres que indica `yyleng`,
      aún cuando la cadena de remplazo sea mucho más grande.
    */
    yytext = remplazo;
    yyleng = tamano;
  }
  ECHO;
}
.|\n ECHO;
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  if (argc != 3) {
    puts("Debe especificarse una palabra, su remplazo y el archivo donde se remplazara
la palabra.\nEjemplo: ./ejercicio5 perro gato entrada.txt");
    exit(1);
  }


  FILE * in = fopen(argv[2], "r");
  if (in == NULL) {
    printf("Fallo al abrirse el archivo '%s'.\n", argv[2]);
    exit(1);
  }
```

```
  /*
    En este archivo colocaremos temporalmente el resultado del remplazo
  */
  FILE * out = fopen(".tmpejer5", "w");
  if (out == NULL) {
      puts("Fallo al abrirse el archivo '.tmpejer5'.");
      exit(1);
  }

  palabra = argv[0];
  remplazo = argv[1];
  tamano = strlen(remplazo);
  yyout = out;
  yyin = in;


  printf("Reemplazando '%s' por '%s' en el archivo %s\n", palabra, remplazo, argv[2]);
  yylex();

  /*
    Al terminar, remplazamos el archivo original de entrada por el archivo temporal.
    El archivo temporal está poblado con los remplazos para este punto.
  */
  rename(".tmpejer5", argv[2]);
}
```

2.6 Ejercicio 6

Elabora un analizador léxico que permita reconocer los componentes léxicos de un programa escrito en pseudocódigo.

```
make ejercicio_6
```

1. Para encontrar las **palabras reservadas** simplemente hicimos expresiones *case-insensitive* donde cada patrón es la palabra reservada en sí. P. ej. `inicio|fin|leer|escribir|si`. Lo mismo sucede con tokens que son literales, como `<=`, `==`, etc.

   Porque la cantidad de palabras reservadas es muy grande decidimos separar la lista de palabras en distintas expresiones en lugar de una sola, esto es solo por la estética del código.

2. `([+-]?{NUMERO})|0`: Para escanear **números** enteros seguimos la regla de buscar un 0, o cualquier número que comience con algun número del 1 al 9, seguido de puros dígitos del 0 al 9.

   `{NUMSIGN}.{DIGITO}+` : Escanear puntos flotantes es similar, solo que se agrega a la expresión un punto, seguido de mínimo un número del 0 al 9.

   `({NUMFLOAT}|{NUMSIGN})[eE][+-]{DIGITO}+`: El caso de los números con notación cientifica varía un poco, primero aclarar que consideramos, por ejemplo `10.0e-02` y `10e-02` como válidos. Entonces, para escanear números escritos en notación cientifica buscamos flotantes/enteros seguidos por una `e` de forma *case-insensitive*, un signo de `+` o `-` y por último una serie de números.

   Para **detectar errores en los números**, más que ponernos a hacer expresiones regulares para cada caso, mejor decidimos que, si en una serie de caracteres todos son del conjunto de caracteres que suelen formar un número, entonces asumimos que de hecho *es un número*, pero mal escrito.

Por ejemplo `'2.3.3'` contiene solo caracteres presentes en un número, pero no es aceptado por ninguna de las expresiones regulares pasadas que describen números bien formulados, por lo tanto, se considera como un error.

3. `[a-z]({NUMCHAR}|_{NUMCHAR})*`: Para recuperar **identificadores** seguimos las reglas dadas:
   - Inicia con una letra del abecedario
   - No puede tener dos `'_'` seguidos, para lograr esto simplemente escaneamos por un solo carácter válido o un `'_'` seguido de un caracter.

     Es así que `'A_A'` es aceptado como válido porque encontramos `'_A'` (`'_'` + letra abecedario o número) mientras que `'A_'`, `'A__'` o `'A__A'` no son válidos porque no le sigue una letra del alfabeto a todos los `'_'` que contienen.

   Para capturar **errores en identificadores** realizamos la misma estrategia que con los números. Si todos los caracteres son del conjunto de caracteres válidos para un identificador, pero no fue aceptado por la expresión que describe identificadores, entonces *debe* de tratarse de un identificador inválido.

4. Para **identificar cadenas** usamos la expresión regular `'((\\')|[^'])*'`, en esta expresión, similar a como hacemos en en el escaneo de identificadores, buscamos o el escape de comilla (`\'`) o cualquier otro carácter que no sea una comilla (`'`), pues se trataría de la comilla de cierre. De esta forma se cubren todos los casos necesarios para capturar las cadenas, el caso de que sea el cierre de la cadena, o una comilla que está *escapada*.

5. El último punto complejo es el de **comentarios multi-linea**. Para escanear correctamente estos usamos un estado `COMENTARIO` que permite manejar de forma especial los caracteres en un comentario, (p. ej. todos los caracteres son válidos).

   De igual forma tomamos en cuenta el caso en que se quiera comentar bloques de codigo que ya contienen comentarios, aunque requiere que el número de tokens de inicio de comentario sean los mismos que el número de tokens de cierre, por ejemplo:

   ```
   (*
     (* Este es un comentario *)
     A = 0;
   *)
   ```

   Para hacerlo incluimos un contador `size_t comentarios_abiertos = 0;` que tiene el nivel actual de comentarios anidados. Cada que encuentra `(*` o `*)` aumenta/disminuye el nivel hasta llegar al nivel final.

   Comentarios del siguiente tipo, con un número no par de *tokens* de inicio/final, son inválidos:

   ```
   (*
     (*
     Este es un comentario
   *)
   ```

```
%{
  #include <stdio.h>
  #pragma GCC diagnostic ignored "-Wunknown-pragmas"
  #pragma warning(disable:4996 6011 6385 4013)
  #pragma GCC diagnostic ignored "-Wunused-function"
  #pragma GCC diagnostic ignored "-Wsign-compare"

  size_t comentarios_abiertos = 0;
%}
```

```
%option noyywrap
%option caseless
%x COMENTARIO

DIGITO [0-9]
NOCERO [1-9]
NUMERO {NOCERO}{DIGITO}*
NUMCHAR [a-z0-9]
NUMSIGN ([+-]?{NUMERO})|0
NUMFLOAT {NUMSIGN}.{DIGITO}+

%%
 /*
   Palabras reservadas.
   Para facilitar la lectura, en lugar de usar una sola expresión las rompimos
   en distintas expresiones.
 */
inicio|fin|leer|escribir|si         { printf("    Reservado: %s\n", yytext); }
entonces|si_no|fin_si|mientras      { printf("    Reservado: %s\n", yytext); }
hacer|fin_mientras|repetir          { printf("    Reservado: %s\n", yytext); }
hasta_que|para|desde|hasta          { printf("    Reservado: %s\n", yytext); }
paso|fin_para                       { printf("    Reservado: %s\n", yytext); }

o|y|no                              { printf("       Logico: %s\n", yytext); }
"<"|"<="|">"|">="|"=="|"="|"<>"     { printf("   Relacional: %s\n", yytext); }
mod|"+"|"-"|"*"|"/"|"**"            { printf("   Aritmetico: %s\n", yytext); }

"("                                 { printf("   Paren abre: %s\n", yytext); }
")"                                 { printf("  Paren close: %s\n", yytext); }
";"                                 { printf("    Separador: %s\n", yytext); }

{NUMSIGN}                           { printf("       Numero: %s\n", yytext); }
{NUMFLOAT}                          { printf("       Numero: %s\n", yytext); }
({NUMFLOAT}|{NUMSIGN})[eE][+-]{DIGITO}+  { printf("       Numero: %s\n", yytext); }
 /*
   Si tiene todos los caracteres de un numero pero no hizo _match_
   con los patrones definidos correctos, entonces debe ser un error.
 */
[0-9+-.E]+                          { printf(" ERROR Numero: %s\n", yytext); }

[a-z]({NUMCHAR}|_{NUMCHAR})*   { printf("Identificador: %s\n", yytext); }
 /*
   Si tiene todos los caracteres de un identificador pero no hizo _match_
   con los patrones definidos correctos, entonces debe ser un error.
 */
({NUMCHAR}|_)+                 { printf("  ERROR Ident: %s\n", yytext); }

'((\\')|[^'])*'                     { printf("       Cadena: %s\n", yytext); }

"#".*                               { printf("   Comentario: %s\n", yytext); }

<COMENTARIO>.                       { ECHO; }
 /*
  Escapamos los saltos de línea de los comentarios multi-linea para hacer
  más amigable la salida mostrada al usuario.
```

```
 */
<COMENTARIO>\n                        { yyleng=2; yytext = "\\n"; ECHO; }
<INITIAL,COMENTARIO>"(*" {
  /*
    Si se trata del primer nivel de comentarios multi-linea entonces cambiamos al
    estado de COMENTARIO
  */
  if (++comentarios_abiertos == 1) {
    BEGIN(COMENTARIO);
    printf("   Comentario: %s", yytext);
  } else {
    /*
      Si no es el primer nivel, entonces ya nos encontramos en el estado de COMENTARIO
    */
    ECHO;
  }
}
<COMENTARIO>"*)"  {
 if (--comentarios_abiertos == 0) {
   /*
    Solo volvemos al estado inicial cuando todos los tokens de inicio de comentarios
    multi-linea han sido resueltos.
   */
   BEGIN(INITIAL); printf("%s\n", yytext);
 } else {
   /*
    En otro caso solo se cerró un nivel anidado más seguimos dentro de un comentario.
   */
   ECHO;
 }
}

\n|" "|\t|\r|\f                     { /* Ignoramos saltos de linea, espacios, etc. */ }
.                                     { printf("ERROR Simbolo: '%s'\n", yytext); }
%%

int main(int argc, char * argv[]) {
  --argc;
  ++argv;

  if (argc != 1) {
    puts("Debe especificarse el archivo del que se leera el pseudocodigo fuente.
\nEjemplo: ./ejercicio6 entrada.txt");
   exit(1);
  }

  FILE * in = fopen(argv[0], "r");
  if (in == NULL) {
   printf("Fallo al abrirse el archivo '%s'.\n", argv[0]);
   exit(1);
  }

  yyin = in;
  yylex();
}
```

3 Referencias

[1] “Lexical analysis with flex, for flex.” Consultado: Sep. 18, 2023. [En Linea]. Disponible en: https://github.com/westes/flex/blob/master/doc/flex.texi