Teoría de la Computación

Grado en Ingeniería Informática

- Prácticas de Laboratorio -

Jordi Bernad Lusilla email: jbernad@unizar.es
Elvira Mayordomo Cámara email: elvira@unizar.es
José Manuel Colom Piazuelo email: jm@unizar.es
Gregorio de Miguel Casado email:gmiguel@unizar.es
Carlos Bobed Lisbona email: cbobed@unizar.es

Dpto. de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Curso 2025-2026

Introducción a las Prácticas de la Asignatura
Entorno de Trabajo y Entrega de Prácticas

Las prácticas de la asignatura Teoría de la Computación abordan aspectos de implementación de reconocedores para expresiones regulares (análisis léxico) y reconocedores de lenguajes caracterizados con gramáticas (análisis sintáctico) mediante las herramientas Unix *Flex* y *Bison*, respectivamente. El aprendizaje de estas herramientas es de interés general, ya que permite abordar, con posterioridad, la construcción de compiladores, programas para la traducción/migración entre formatos de ficheros y similares.

Entorno de Trabajo

Las prácticas de Teoría de la Computación se realizarán en los ordenadores de los laboratorios asignados sobre el servidor de docencia *lab000*. Para conectarse a *lab000* desde un ordenador del laboratorio es preciso autentificarse con un usuario y contraseña específico para *lab000*. El usuario está compuesto de la letra `a` seguida de su NIP, `a<NIP>` (por ejemplo, `a642178`), y la contraseña, en caso de que la desconozca, tiene que crearla rellenando el formulario disponible en:

`https://webdiis.unizar.es/cgi-bin/hendrix-newpw`

Para conectarse en remoto desde su propio ordenador se deberá utilizar la dirección **lab000.cps.unizar.es** utilizando algún cliente ssh (ej., MobaXterm[1], Putty, etc).

Recuerde que se puede trabajar en local, esto es, sobre sus propios ordenadores, pero antes de entregar la práctica se tendrá que probar el código en lab000. **En esta máquina se harán las correcciones de las prácticas y deberá comprobar que su código compila y funciona correctamente en lab000.**

Para usar Flex y Bison en el propio ordenador se necesita tener instalado:

- El compilador del lenguaje C `gcc` junto con las librerías de Flex y Bison.
- Compilador de Flex.

[1] `https://mobaxterm.mobatek.net/`

- Compilador de Bison.

Usuarios de Linux. En las distribuciones más habituales del sistema operativo Linux se puede instalar Flex y Bison ejecutando desde un terminal los comandos[2] `sudo apt-get install flex` y `sudo apt-get install bison`. Por otro lado, se ha detectado que algunos usuarios de la distribución Linux Mint tienen problemas al compilar con `gcc` los ficheros generados por Flex. Para resolver este problema en Linux Mint se debe instalar Flex con la siguiente orden: `sudo aptitude flex libfl-dev`.

Usuarios de MacOs. Los ordenadores que tienen instalada alguna de las distribuciones más comunes del sistema operativo MacOs ya tienen instaladas las tres herramientas necesarias para realizar las prácticas de la asignatura. En este caso, para compilar y ejecutar los programas creados con Flex y Bison se tendrá que introducir desde un terminal los comandos que posteriormente se detallarán.

Usuarios de Windows. Para los usuarios de Windows, se tiene disponible en moodle en la sección de "Prácticas de Laboratorio" el archivo *instalarFlexBisonWin.zip* con todo lo necesario para poder compilar las prácticas. Las instrucciones de instalación son:

- El archivo *instalarFlexBisonWin.zip* contiene dos carpetas comprimidas: `MinGW` (compilador gcc y librerías de flex y bison); `winflexbison` (commpiladores de flex y bison). Guardar estas dos carpetas en la raíz del volumen `C`, esto es, en `C:\`
- Añadir al `PATH` del usuario los directorios `C:\MinGW\bin` y `C:\winflexbison`. El objetivo de este paso es poder compilar los programas desde cualquier directorio del ordenador. Para cambiar el `PATH` del usuario ir a "Panel de Control→Sistema→Configuración avanzada del sistema→Variables de entorno" y añadir a la variable `PATH` del usuario los dos directorios nombrados anteriormente. Se tiene disponible en moodle un vídeo explicando este paso[3] .
- Ejecutar la consola de comandos, "Símbolo del sistema". Para abrir esta consola, se tendrá que buscar el programa "cmd" en el ordenador y ejecutarlo.
- En la consola se ejecutarán los comandos que posterioremente se especifican para compilar y ejecutar los programas creados con Flex y Bison.
- Si tiene alguna duda o problema, por favor, consulte con su profesor de prácticas.

[2] En distribuciones basadas en Debian, en otras hay que utilizar el gestor de paquetes apropiado.

[3] De otros años, puede que os colisione con algún otro compilador que hayáis instalado en anteriores asignaturas. Para solucionarlo, las variables de entorno de los nuevos elementos tienen que tener prioridad sobre las anteriores - aparecer antes - y en el PATH debe estar la ruta de nuestro mingw antes del anteriormente instalado.

Edición de programas. Independientemente del sistema operativo del ordenador, la edición de los programas fuente de Flex y Bison se realizará con un editor de texto plano, como por ejemplo, gedit, sublime, notepad, notepad++, vim, etc.

Material de consulta para flex y bison. En las siguientes URLs se puede encontrar todo lo relacionado a estas herramientas:

- http://flex.sourceforge.net/
- http://www.gnu.org/software/bison/

También se utilizarán como material de referencia el siguiente libro y manuales:

- *flex & bison*, John Levine, Ed. O'Reilly.
- Manual de flex versión 2.5 (disponible en moodle)
- Manual de bison versión 1.27 (disponible en moodle)

Entrega de Prácticas

Las prácticas se realizarán en **parejas**. Los dos miembros de la pareja deberán **estar apuntados al mismo grupo de prácticas**.

Para cada una de las prácticas se deberá entregar un fichero *.zip* que contenga una memoria en formato *PDF* y los ficheros fuente y de prueba para cada ejercicio planteado.

Los siguientes apartados detallan los contenidos de la memoria, el procedimiento para empaquetar en un fichero *.zip* los ficheros para la entrega y finalmente el mecanismo de entrega a través de Moodle.

El incumplimiento de las normas establecidas en este apartado para el formato de la memoria y/o ficheros se reflejará en la calificación de la práctica.

Las copias o plagios que se detecten en las memorias y/o programas supondrán la apertura de un expediente sancionador conducido al suspenso directo de la parte práctica de la asignatura.

Formato de la memoria

- **Portada:** Número de Práctica, Grupo de prácticas, Autores y NIPs. Ejemplo:

Grupo Miércoles 12:00-14:00 semanas B
– Práctica 1 –
Autor: Al Anturing
NIP:123456
Autor: Ku Rtgodel
NIP:654321

- **Una sección para cada ejercicio resuelto.** Razone todas las decisiones de implementación que ha tomado en la elaboración de su código e incluya las pruebas de ejecución realizadas. Ejemplo:

Ejercicio 1:

1. Resumen

He creado el patrón X para poder reconocer Y

...

2. Pruebas

Para la entrada Z he obtenido la salida W

...

Nota: el formato del fichero de la memoria deberá ser *PDF*.

Empaquetado y entrega de los ficheros

- Verifique que todos sus ficheros fuente (*.l* de *Flex* y *.y* de *Bison*) contienen en sus primeras líneas número de práctica y ejercicio, así como los NIPs y nombres de los autores. Todos los programas deberán estar debidamente documentados.
- Verifique que los ficheros que va a presentar compilan y ejecutan correctamente en lab000.
- Cree un fichero comprimido .zip llamado

`nip`PrX`.zip`

donde *nip* es el identificador personal y *X* es el número de práctica (1, 2, 3 ó 4). Este fichero comprimido **debe contener exclusivamente** el fichero con la memoria en formato *PDF*, los ficheros fuente con su código (*.l* de *Flex* y *.y* de *Bison*) y los de prueba (*.txt* de texto). No usar subdirectorios.

- **Uno y solo uno de los miembros de la pareja** enviará el fichero zip mediante un enlace a una tarea de moodle disponible en la sección "Prácticas de Laboratorio". Mediante este enlace se entregará el fichero `nipPrX.zip`

Importante:

- **La fecha límite de entrega para cada una de las prácticas será hasta el día anterior (incluido) a la sesión en la que comience la siguiente práctica. Para la última práctica se concretará una fecha específica.**
- **Es necesario estar apuntado en un grupo de prácticas para poder entregar las prácticas**. En caso de no estar apuntado a ningún grupo de prácticas se entenderá que el alumno ha decidido realizar el examen de prácticas para obtener la nota de prácticas de la asignatura. Para apuntarse a un grupo de práctica se debe consultar el enlace "Elección de grupo de prácticas" disponible en moodle.

Práctica 1: Introducción a Flex

Tareas

1. Aprender a acceder y trabajar con la cuenta en lab000.
2. Aprender a trabajar localmente en el ordenador.
3. Leer las págs 1-4 del libro *flex & bison*, John Levine, Ed. O'Reilly.
4. Leer la introducción de esta práctica y realizar los ejercicios 1 a 4 propuestos en la parte A (primera sesión).
5. Leer la introducción de la parte B y realizar los ejercicios propuestos en la parte B (segunda sesión).
6. Elabore la memoria de la práctica y entréguela junto con los ficheros fuente **de la parte B** según el Procedimiento de Entrega de Prácticas explicado en la Introducción a las Prácticas de la Asignatura (página 4 de este documento). La práctica 1 tiene 2 sesiones. La fecha tope de entrega será hasta el día anterior al comienzo de la Práctica 2 (tercera sesión).

Nota: El incumplimiento de las normas de entrega se reflejará en la calificación de la práctica.

Introducción

El objetivo principal de esta práctica de la asignatura es familiarizarse con la herramienta de creación de analizadores léxicos *Flex*. Para ello, se propone la creación con dicha herramienta de una serie de pequeños procesadores de texto.

Para compilar y ejecutar los programas creados con flex, hay que usar los comandos descritos en la Figura 1.1[4].

El alfabeto Σ que maneja *Flex* está formado por los caracteres manejables por el sistema (p. ej. todos los símbolos del código ASCII). En las prácticas se trabaja con

[4]En sistemas MacOS, si da problemas al enlazar, sustituir la opción -lfl por -ll, esto es, compilar con `gcc lex.yy.c -ll -o nombre_ejecutable`.
Por otra parte, en los sistemas Windows es necesario que al compilar con `gcc lex.yy.c -lfl -o nombre_ejecutable` la opción `-lfl` aparezca después de `lex.yy.c`

```
/> flex nombre_fichero_fuente.l
/> gcc lex.yy.c -lfl -o nombre_ejecutable
/> ./nombre_ejecutable < fichentrada > fichsalida
```

Figura 1.1: Comandos para compilar y ejecutar con Flex.

letras (distinguiendo entre mayúsculas y minúsculas, no se usa la ñ), números, 8 signos de puntuación (! , . : ; ( ) ?), 3 separadores (espacio, tabulador y salto de línea) y 23 símbolos visibles ($" \# \$ \% \& ' * + - / < = > @ [\ ] \hat{} \_ \{ | \} \sim$). No es necesario preocuparse de símbolos no visibles, letras acentuadas o cualquier otro carácter que no hayamos mencionado aquí. Además, **supondremos que los ficheros de texto siempre terminan con un final de línea**.

Trabajando con Flex

Supongamos que tenemos instalado en nuestro ordenador un molesto virus que al pulsar la tecla 'a' introduce dos aes. Nuestro antivirus no es capaz de detectar el virus y tenemos prisa por enviar correctamente el siguiente mensaje,

`Vaayaa problemaa`

Aplicando los conocimientos adquiridos en el curso pasado, para salvar la situación, nos planteamos hacer un pequeño programa en lenguaje C que sustituya cada aparición de 'aa' por 'a', y de este modo obtener el mensaje correcto,

`Vaya problema`

Con la herramienta Flex podemos crear este tipo de programas con suma facilidad. Nos permite centrarnos en describir los patrones que buscamos (en nuestro caso, `aa`), y una vez encontrado aplicar alguna acción (escribir solo una `a`), sin necesidad de preocuparnos por crear código para la apertura/cierre de ficheros, bucles para leer, etc. El único código que tendríamos que escribir para resolver el problema utilizando Flex sería,

```
%%
aa    {printf("a");}
%%
```

Una vez creado un fichero de Flex con el contenido anterior, utilizaremos el compilador de Flex para generar un fichero con el código necesario en lenguaje C (por defecto, este fichero de C se denomina `lex.yy.c`) que lea un fichero y modifique dos aes por una a. Finalmente, mediante el compilador de C `gcc` crearemos un ejecutable. En definitiva,

la herramienta Flex genera programas escritos en C a partir de un fichero en formato Flex capaz de escanear ficheros de texto buscando patrones y aplicando acciones tras haberlos encontrado (ver Figura 1.2). Los ejecutables creados mediante Flex se denominan *escáneres léxicos* o *analizadores léxicos*.

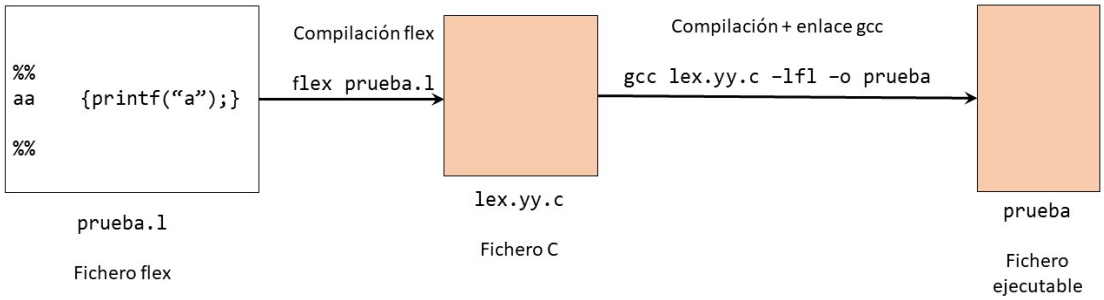

Figura 1.2: Flujo de trabajo con Flex.

Formato de ficheros Flex

Un fichero en formato Flex se divide en tres secciones separadas por los caracteres '%%' **escritos al inicio de una línea**. Estas secciones se llaman: sección de declaraciones, reglas y código:

```
  declaraciones
%%
   reglas
%%
  codigo C
```

Sección declaraciones. Como su propio nombre indica esta sección es utilizada para realizar declaraciones. Por ejemplo, se puede usar para declarar variables en C, incluir librerías, o cualquier código de C. A lo largo de las clases de prácticas se explicarán otro tipos de declaraciones interesantes como definiciones de nombres de patrones y declaración de condiciones de arranque. Por el momento, veamos cómo usar esta sección para introducir declaraciones en C. Suponga que desea contar el número de apariciones de 'aa'. El siguiente código resuelve el problema,

```
%{
  #include <stdio.h>
  int contador = 0;
%}
%%
aa    {contador++;}
%%
```

Observe que se ha definido un bloque de código C en la sección de declaraciones: todo lo que se encuentra entre `%{` y `%}`, ambos escritos al **inicio de una línea**. En este bloque podemos escribir cualquier código en C. Habitualmente lo utilizaremos para incluir librerías y declarar e inicializar variables de C.

Sección de reglas. Una regla se compone de un patrón y una acción. Un patrón está definido mediante una *expresión regular* y una acción es una serie de instrucciones escritas en C. En el ejemplo anterior, la sección de reglas solo contiene una regla compuesta por la expresión regular `aa` y la acción es aumentar en una unidad la variable `contador` declarada en la sección de declaraciones. En las próximas sesiones se explicará cómo formar expresiones regulares en Flex más complejas que nos permitan identificar patrones más sofisticados.

Para que una regla esté bien formada:

- la expresión regular **debe estar escrita al inicio de una línea**;
- tras la expresión regular, se escribirán uno o más espacios en blanco, y a continuación, se escribirá la acción. La acción puede estar delimitada por `{` y `}`, en cuyo caso podremos escribir las instrucciones de la acción repartidas en varias líneas, o si no hay llaves, la acción se compone de las instrucciones que aparecen en la misma línea que la expresión regular. Los dos siguientes códigos realizan la misma tarea:

```
%{
  int contador = 0;
%}
%%
aa    {contador++;
       printf("a");
      }
%%
```

```
%{
  int contador = 0;
%}
%%
aa    contador++; printf("a");
%%
```

Sección de código. En la última de las secciones podemos escribir cualquier código en C. Esta sección se añadirá al fichero `lex.yy.c` generado por el compilador de Flex tal cual la escribamos. Considere el siguiente código para contar y mostrar el número de veces que aparece `aa`:

```
%{
  #include <stdio.h>
  int contador = 0;
%}
%%
aa   {contador++;}
%%
int main() {
  int error = yylex();
  if (error == 0) { //si escaner finaliza correctamente
    printf("Total de aa: %d\n", contador);
  }
  return error;
}
```

En la sección de código hemos declarado una función: la función `main` que todo programa en C debe implementar. En la primera línea de código del `main` se llama a la función `yylex`. `yylex` es una función definida e implementada por Flex encargada de lanzar el analizador léxico. Esta función analiza con las reglas definidas en la sección anterior el fichero de entrada y devuelve un número con un código de error. Si este número es igual a 0, significa que el analizador ha terminado correctamente el escaneo. En caso contrario, implica que se ha producido algún problema, como por ejemplo, no tener permiso de lectura sobre el fichero escaneado o producirse algún otro tipo de error de lectura/escritura. La segunda línea de la función `main` comprueba que el código de error sea 0, y en ese caso, muestra con `printf`[5] el valor de la variable `contador`. Finalmente, devuelve el código de error generado por `yylex`.

Si la sección de código está vacía como en otros ejemplos anteriores, Flex crea por defecto una función `main` similar a:

```
int main() {
  int error = yylex();
  return error;
}
```

[5] `printf` es una función de C para mostrar datos mediante lo que se llama una *cadena formateada*. Para más información sobre esta función, consultar https://en.wikipedia.org/wiki/Printf_format_string

esto es, si la sección de código no contiene nada, simplemente se lanza el analizador léxico y se devuelve el código de error.

Cómo funciona el analizador léxico generado

Consideremos el siguiente código de Flex,

```
%{
  int contador = 0;
%}
%%
aa    {contador++; printf("a");}
%%
int main() {
  int error = yylex();
  if (error == 0) {
    printf("Total␣de␣aa:␣%d\n", contador);
  }
  return error;
}
```

Supongamos que analizamos la siguiente entrada con el programa generado por Flex:

```
Vaayaa␣problemaa
```

En el momento de llamar a la función `yylex()` dentro de `main`, se empieza a escanear el texto.

```
Vaayaa␣problemaa
↑
```

La flecha hacia arriba indica donde se encuentra el puntero de lectura del texto, esto es, cuál es el siguiente carácter que se leerá de la entrada. Empezamos en la `V`. Flex (el analizador léxico generado por Flex) busca si desde ese punto puede aplicar alguna expresión regular entre las reglas de las que dispone. Como no puede aplicar ninguna regla, la única es `aa`, el comportamiento por defecto de Flex es mostrar en la salida estándar el carácter y avanzar el puntero de lectura al siguiente carácter (consume de la entrada un carácter y avanza):

Entrada

```
Vaayaa␣problemaa
␣↑
```

Salida

```
V
```

En este punto Flex sí puede aplicar una regla, en este caso `aa`. Consume de la entrada todos los caracteres que concuerdan con la expresión regular y aplica la acción asociada: consume de la entrada los caracteres `aa`, avanza el puntero sobre la '`y`', muestra en la salida una `a`, y suma uno a `contador`:

Entrada

```
Vaayaa␣problemaa
␣␣␣↑
```

Salida

```
Va
```

En estos momentos el puntero de lectura se encuentra sobre la '`y`' y el analizador se comporta de la misma forma que la descrita anteriormente hasta consumir todos los caracteres de la entrada. Note que Flex trata el carácter espacio en blanco y salto de línea ('`\n`') de igual forma que cualquier otro carácter imprimible:

Entrada

```
Vaayaa␣problemaa
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣↑
```

Salida

```
Vaya problema
```

El analizador lanzado por la función `yylex` termina correctamente, y finalmente se ejecuta la línea `printf("Total de aa: %d\n", contador)`, mostrando el resultado definitivo:

Entrada

```
Vaayaa␣problemaa
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣↑
```

Salida

```
Vaya problema
Total de aa: 2
```

Reglas en colisión

Tomemos en consideración ahora las siguientes reglas de Flex:

```
 . . . .
%%
aa    {contador1++;}
aab   {contador2++;}
%%
 . . . .
```

y supongamos que nos encontramos analizando la entrada en la siguiente situación:

```
     aabaa
     ↑
```

Tenemos dos reglas que se pueden aplicar: `aa` y `aab`. Por defecto, Flex elige siempre aplicar la regla que consuma más caracteres de la entrada, en este caso `aab`. Por tanto, se sumaría uno a `contador2` y el puntero para la siguiente lectura pasaría a estar sobre el carácter '`a`' tras la última '`b`' consumida:

```
aabaa
  ↑
```

El comportamiento de Flex es voraz con respecto a la entrada. Siempre aplica la regla que consume más datos de la entrada, o en otras palabras, aplica la regla con la que concuerdan más caracteres. Cuando dos o más reglas se pueden aplicar, Flex se comporta de la siguiente forma:

- Aplica la regla que más caracteres consume de la entrada.
- En caso de que varias reglas consuman el mismo número de caracteres, aplica la regla que aparece antes en el fichero fuente de Flex.

Funciones y variables definidas en Flex

Flex define un conjunto de funciones y variables que nos ayudan en la implementación de los analizadores. La función `yylex` es un ejemplo de función definida por Flex, que como ya se ha explicado, se encarga de lanzar el escáner. Aunque existen una gran variedad de funciones y variables definidas por Flex, destacamos estas dos variables por su utilidad a la hora de implementar las prácticas de la asignatura:

- `yytext`: la variable `yytext` es un array de caracteres donde se guarda el último patrón encontrado.
- `yyleng`: es una variable entera que almacena el número de caracteres de `yytext`.

Últimas consideraciones

Como ya se ha indicado, muchos de los elementos propios de Flex, como los cambios de sección con `%%` o las expresiones regulares de las reglas, se deben escribir al inicio de una línea. Como regla general, si una línea empieza por un espacio en blanco, el compilador de Flex entenderá que el resto de la línea es código escrito en C para introducir tal cual en el fichero generado `lex.yy.c`. Por tanto, solo las líneas que empiezan por un carácter distinto del espacio en blanco pueden tener un sentido especial para el compilador de Flex. El resto se consideran líneas que contiene código en C. Puede probar a introducir un espacio en blanco delante del patrón `aa` del ejemplo anterior. El compilador de Flex no informará de ningún error y generará el fichero de C `lex.yy.c`. Sin embargo, al compilar este fichero con gcc aparecerá un error diciendo que el tipo de dato `aa` no ha sido definido.

Esto es, Flex, en lugar de considerar `aa` como un patrón lo ha considerado como código en C a insertar en `lex.yy.c`.

Pero también, por regla general, se cumple lo contrario: si una línea empieza por un carácter distinto de blanco, entonces tiene un sentido especial para Flex. Observe el siguiente error muy común al entregar una práctica por parte del alumnado. Tras implementar, ejecutar y probar el código de un práctica, justo antes de entregarla, los alumnos Fulanito y Menganito introducen sus datos personales en un comentario de C estilo línea,

Incorrecto

```
// Autores: Fulanito  NIP: 432156 ; Menganito NIP; 767543
%%
aa    {printf("a");}
%%
```

El profesor compila con Flex la práctica y se encuentra con un error: el compilador de Flex no sabe qué hacer con `//` al inicio de una línea. El problema se soluciona simplemente introduciendo un espacio en blanco delante de `//`: ahora Flex entiende que es código en C que debe introducir en el fichero `lex.yy.c` y no produce error alguno.

Correcto

```
  // Autores: Fulanito  NIP: 432156 ; Menganito NIP; 767543
%%
aa    {printf("a");}
%%
```

Parte A: Introducción al manejo de *Flex*

Ejercicio 1

Disponemos de un fichero de texto con las cuentas de mail de una serie de usuarios. Escribe un programa con *Flex* de nombre *ej1.l* que sustituya cualquier correo de *hotmail* por *gmail*. Esto es, cada vez que aparezca la cadena *@hotmail* la cambie a *@gmail*

Ejemplo:

Entrada:

```
perico@hotmail.com  ana@unizar.es
sab@hotmail.com  javier@garcia.com
maria@hotmail.com  modrego@rm.edu
jose@unizar.es
```

Salida:

```
perico@gmail.com  ana@unizar.es
sab@gmail.com  javier@garcia.com
maria@gmail.com  modrego@rm.edu
jose@unizar.es
```

Ejercicio 2

Elabora un programa en *Flex* de nombre *ej2.l* que permita contar el número de usuarios de correo de hotmail, esto es, el número de apariciones de la cadena *@hotmail*

Ejemplo:

Entrada:

```
perico@hotmail.com  ana@unizar.es
sab@hotmail.com  javier@garcia.com
maria@hotmail.com  modrego@rm.edu
jose@unizar.es
```

Salida:

```
perico@hotmail.com  ana@unizar.es
sab@hotmail.com  javier@garcia.com
maria@hotmail.com  modrego@rm.edu
jose@unizar.es
Total de usuarios: 3
```

Ejercicio 3

Escribe un programa con *Flex* de nombre *ej3.l* que sustituya cualquier email de la Universidad de Zaragoza por un correo del mismo usuario de gmail. Es decir, se debe sustituir cualquier aparición de *@unizar.es* por *@gmail.com*

Razona las siguientes preguntas:

- Con lo que te han explicado en la clase de prácticas, ¿qué ocurre si un usuario tiene como cuenta de correo *jrg@unizaroes.es*? ¿Se modifica adecuadamente el mail? ¿Por qué?
- Intenta entender las páginas 19 y 20 del libro *flex & bison* e indica dos formas distintas de solucionar el problema. Cambia el código del fichero *ej3.l* con una de las dos formas propuestas.

Ejercicio 4

Construye un programa en *Flex* de nombre *ej4.l* que modifique:

- todas las apariciones de un cifra (del 0 al 9) por el número siguiente al que representa la cifra;
- todas las apariciones de un salto de línea por dos saltos de línea.

Ejemplo:

Entrada:

```
Mi numero de telefono es 548271210
ponte en contacto con
el asistente 59 en 04 minutos.
```

Salida:

```
Mi numero de telefono es 659382321

ponte en contacto con

el asistente 610 en 15 minutos.
```

Observaciones:

- Recuerda que se puede usar la función *atoi* para transformar una cadena de caracteres en un número entero.

```
int n = atoi(s);
```

- Para escribir un número entero con *printf* se puede usar

```
printf("%d", n);
```

Parte B: Introducción al manejo de *Flex*

Introducción

El objetivo principal de esta parte es familiarizarse con la herramienta de creación de analizadores léxicos *Flex* y **aprender aspectos básicos sobre teoría de lenguajes y expresiones regulares**. Para ello, se propone la creación con dicha herramienta de una serie de pequeños procesadores de texto.

El cuadro 1.1 muestra algunas correspondencias entre la notación empleada en clase de teoría y la que se utiliza en *Flex* para trabajar con expresiones regulares (ERs).

Operación	**ERs Teoría**	**ERs Flex**
Concatena ERs	$\cdot$ ó {vacío}	{vacío} (yuxtaposición)
Unión de ERs	$+$	|
Cerradura de Kleene	*	$*$
Cerradura Positiva	$^+$	$+$
Paréntesis	( )	( )
Símbolos a,b,c,d,0,1,2	$\{a, b, c, d, 0, 1, 2\}$	$[abcd012]$ ó $[a-d0-2]$
Cadena vacía ó a	$\epsilon + a$	$a?$
Cadena vacía	ϵ	Sin equivalencia (ver $*$ ó ?)
Conjunto vacío	$\emptyset$	Sin equivalencia

Cuadro 1.1: Correspondencia entre notaciones

Ejercicio 5

Los ficheros en formato *csv* (comma separated value) se componen de líneas de texto formadas por grupos de valores separados por comas. Para simplificar el problema, supondremos que un valor se compone de un conjunto de letras mayúsculas, minúsculas, dígitos, el carácter ':', y espacios en blanco; **pero no puede empezar ni acabar por espacio**. Por tanto, con esta definición, para resolver el problema podemos suponer que los valores no son vacíos (no están compuestos por cero o más espacios en blanco), y que los espacios al inicio y al final no forman parte del valor (aunque sí son caracteres del fichero).

Implementa un analizador léxico en *Flex* (fichero *ej5.l*) que analice un fichero de texto en formato *csv* y genere otro con la siguiente información:

- cuántos caracteres tiene el fichero (excepto saltos de línea), cuántas lineas de texto y, de éstas, cuántas están vacías. Supondremos que el fichero siempre termina con un final de línea;
- cuántos valores contiene el fichero;
- total de caracteres del valor de mayor longitud;
- total de valores que son duplas de números separados por : (e.j., 123:31)
- total de valores que no tienen ningún dígito.

La salida estará compuesta por exactamente seis líneas de texto con el formato:

```
C:<total de caracteres excepto saltos de líneas>
L:<total de líneas>
LV:<total de líneas vacías>
V:<total de valores>
M:<total de caracteres del valor más largo>
D:<total de duplas numéricas>
N:<total de valores sin ningún dígito>
```

Ejemplo (en este ejemplo, marcamos los espacios en la entrada con ␣):

Entrada:

```
Ciudad␣,:Habitantes:,Comida,Otros¶
Zaragoza,␣600000,␣migas,1:10¶
␣␣¶
Barcelona,1000000,pan␣y␣tomate¶
Huesca,30000,trenza␣de␣Huesca,3:123¶
```

Salida:

```
C:128
L:5
LV:1
V:15
M:16
D:2
N:10
```

Ejercicio 6

Muchas órdenes de linux[6], como *egrep, grep, find, sed, etc*, tienen como parámetro de entrada una expresión regular. Por ejemplo, la orden *egrep*, en su versión más simple, tiene dos parámetros: el primero es una expresión regular entre comillas simples; el segundo es el nombre de un fichero de texto. Al ejecutar la orden, muestra aquellas líneas del fichero que concuerdan con el patrón de la expresión regular. Por ejemplo, si tenemos el fichero *texto.txt*, al ejecutar la orden *egrep 'la' texto.txt* nos mostrará por pantalla aquellas líneas del fichero que contienen 'la'.

Fichero texto.txt

```
hola, que tal estas?
estoy en casa
de la familia Costa
saludos
```

```
labo00:$ egrep 'la' texto.txt
hola que tal estas?
de la familia Costa
lab000:$
```

Escribe en un fichero llamado *ej6.txt* cada una de las órdenes necesarias para mostrar por pantalla:

1. Las líneas de un fichero llamado *t.txt* que **no** empiecen y finalicen con un dígito.
2. Las líneas de *t.txt* que contengan un identificador, una dirección IP y un alias separados por espacios en blancos (al inicio y al final de la línea también pueden aparecer espacios en blanco). Un identificador y un alias serán secuencias de letras y cifras que **empiecen por una letra**, mientras que consideraremos IPs candidatas a las secuencias compuestas por 4 números de uno a tres dígitos separados por puntos (por ejemplo, 127.0.0.1).

   Para la siguiente entrada, egrep sólo debería mostrar la segunda línea:

```
␣␣fermin37␣␣25.255.21.4␣¶
␣␣fermin37␣␣25.255.21.4␣pepe¶
␣␣37fermin␣␣25.255.21.4␣pepe␣¶
␣␣fermin37␣␣25.255.21␣¶
␣␣fermin37␣␣25.255.21.8.67␣¶
␣␣fermin37␣␣25.255..21.8␣¶
␣␣fermin37␣␣25.2556.1.8␣¶
␣␣fermin37␣␣25.256.1.8␣32fermin␣¶
```

[6] Las expresiones regulares son unas grandes amigas de los administradores de sistemas Linux. Todo lo que ocurre en una máquina Linux queda registrado en ficheros log, ficheros de texto donde se guarda información sobre distintos eventos: usuarios conectados, envío de mailes, direcciones IP de máquinas que intentan acceder al sistema, etc. Para filtrar esta información, las expresiones regulares son un herramienta básica que todo buen administrador debe dominar.

3. Las líneas de *t.txt* que contengan un número impar en decimal. Consideraremos que los números están formados por la mayor cantidad posible de dígitos consecutivos. Por ejemplo, la línea *"hola 22dados"*, contiene un único número, *22*, y por tanto, no se debe mostrar; sin embargo, *"131hola 212dados"*, sí debe mostrarse ya que contiene dos números, *131* y *212*, y uno de ellos es impar.

El fichero *ej6.txt* a presentar contendrá únicamente cuatro líneas de texto:

- Una primera línea con los datos personales de nombres y NIPs de los autores.
- Tres líneas con la solución a los apartados 1-3, respectivamente. Cada línea tendrá el formato:

```
egrep 'expresion_regular_propuesta' t.txt
```

Observaciones:

Para evitar problemas de cómo representar el carácter '\t' (tabulador) con egrep en una expresión regular, no se considerará la aparición de tabuladores en los ficheros.

Se recuerda que los comandos `egrep` utilizados para resolver el ejercicio deberán ejecutarse correctamente en lab000. Por ejemplo, la expresión regular en,

```
egrep '[0-9]{1,3}' t.txt
```

no se ejecutará como se espera en lab000 (líneas que contengan de 1 a 3 dígitos consecutivos) ya que el comando `egrep` en lab000 no considera que los carácteres '`{`' y '`}`' tengan un significado especial. Este comportamiento puede ser distinto en otras máquinas. Nótese que usando convenientemente el operador `?` (0 o 1 repetición) se puede simular el comportamiento de `{1,3}`.

Ejercicio 7

Figura 1.3: Cuadrícula. Posición final tras $IDDAADF$

Tenemos un robot que se puede mover por una cuadrícula tal y como se aprecia en la Figura 1.3. El robot empieza sus movimientos siempre en la posición $(0, 0)$ de la cuadrícula y se puede mover hacia la derecha y hacia arriba. Una secuencia de movimientos del robot se especifica por una cadena que empieza por B (inicio el robot), termina por E (fin de robot), y se compone de R (derecha, hacia el este) y U (arriba, hacia el norte). Por ejemplo,

$$BRRUURE$$

dejaría al robot en la posición $(3, 2)$ de la cuadrícula (Figura 1.3).

Generar un fichero en *Flex* llamado *ej7.l* que modifique cualquier aparición de cadenas que empiezen por B, finalicen por E y el resto de caracteres sean R y U:

- si la cadena deja al robot en una posición (n, m) con m un múltiplo de 3, se pondrá un '++' al inicio y final de la cadena
- si la cadena deja al robot en una posición (n, m) con m un número par que no sea múltiplo de 3, se pondrá un ':' al inicio y final de la cadena
- si la cadena deja al robot en una posición (n, m) con m un número impar no múltiplo de 3, se pondrá un '−' al inicio y al final de la cadena.

Supondremos que en el texto de entrada las únicas letras mayúsculas que aparecen son I, D, A y V. Ejemplo:

Entrada:	**Salida:**

`querido francisco:`
`me puedes marcar como hemos quedado las cadenas`
`BE, BRUURUUE,`
`BUURURRRUUURRRRRE`
`BUUUUUUUUUE BRUE`
	`querido francisco:`
	`me puedes marcar como hemos quedado las cadenas`
	`++BE++, :BRUURUUE:,`
	`++BUURURRRUUURRRRRE++`
	`++BUUUUUUUUUE++ −BRUE−`

ATENCIÓN: No se permite el uso de variables contadores para resolver el problema.