Memoria Práctica `Lex`
Modelos de Computación

Joaquín Avilés de la Fuente Arturo Olivares Martos

27 de diciembre de 2024

Índice

Resumen

En la presente memoria se detallará la práctica llevada a cabo en la asignatura de Modelos de Computación del tercer curso del Doble Grado en Ingeniería Informática y Matemáticas de la Universidad de Granada. En ella, hemos implementado un menú con diversas funciones para analizar un texto introducido por el usuario, el cual detallaremos más adelante en las distintas secciones de la memoria.

1. Introducción

En nuestro caso, para no limitarnos a un único caso de uso, lo cual probablemente implicaría usar menos expresiones regulares, hemos decidido implementar un menú con diversas opciones que permiten al usuario analizar un texto introducido por él mismo. Esto proporciona así más versatilidad, a la vez que nos facilita la implementación de un mayor número de expresiones regulares.

La funcionalidad del programa se puede probar de forma directa mediante el menú creado, o se puede tambiar probar por separado con cada programa. Además, hemos creado un archivo `makefile` que permite que la ejecución sea más sencilla.

Todos estos aspectos los explicaremos de forma detallada en cada una de las secciones de este artículo.

Observación. Todos las rutas relativas que se proporcionan suponen que nos encontramos en la carpeta raíz de esta práctica.

2. Comprobación de Números de Teléfono

Este analizador se encuentra en el archivo `Telefonos/regex_tfno.l`. Para compilarlo, se debe ejecutar el siguiente comando:

```
$ flex++ -o regex_tfno.cpp regex_tfno.l
$ g++ -Wall -o regex_tfno regex_tfno.cpp -lfl
```

Para ejecutarlo, se debe ejecutar el siguiente comando:

```
$ ./regex_tfno <fichero>
```

donde `<fichero>` es el archivo que se desea analizar en busca de números de teléfono. De forma alternativa, se recomienda usar el `makefile` proporcionado usando la siguiente regla:

```
$ make telefonos
```

En este caso, el archivo de texto que se analizará es `Telefonos/telefonos.txt`, por lo que deberán almacenarse en ese archivo las cadenas que se deseen procesar.

Este programa analiza el archivo pasado en búsqueda de números de teléfono. Por un lado, informa de las cadenas que no han sido identificadas con números de teléfono de ningún país registrado y, por otro lado, de los números de teléfono válidos que se han encontrado, informa en qué país son válidos. Actualmente tan solo se ha implementado la detección de números de teléfono españoles, puesto que son los que conocemos de primera mano, pero la extensión a otros países es directa. Para detectar los números de teléfono, se han usado las siguientes expresiones regulares:

- `tfno_espanol_grupos3`:

  ```
  ((00|"+")34(" ")?)?([0-9]{3}(" ")?){3}
  ```

```
arturoolvrs@arturoolvrs-MacBookAir:~/Desktop/Lex-Practica$ make telefonos
flex++   Telefonos/regex_tfno.l
mv lex.yy.cc Telefonos/regex_tfno.cpp
g++   -std=c++11 -g -Wall   Telefonos/regex_tfno.cpp -o Telefonos/regex_tfno_exe -lfl
./Telefonos/regex_tfno_exe Telefonos/telefonos.txt


Cadenas leidas no identificadas:
        - 12345678912
        - abcdefgh

Números por Pais:
- España:
        - +34 888 88 88 88
        - 0034 999999999
        - 643 333 333
        - 999 999 997

arturoolvrs@arturoolvrs-MacBookAir:~/Desktop/Lex-Practica$
```

Figura 1: Ejemplo de ejecución del programa `regex_tfno`.

En este caso, se da la posibilidad a que el número de teléfono esté agrupado en grupos de 3 cifras, separados por un espacio. En primer lugar, puede comenzar con el prefijo internacional (`34` en el caso de España), que ha de ir precedido de `00` o `+`. A continuación, se encuentras los tres grupos de 3 cifras, en total 9 cifras. Se permite que haya un espacio entre los grupos de cifras y tras el prefijo internacional.

- `tfno_espanol_grupos2`:

  ```
  ((00|"+")34(" ")?)?([0-9]{3}(" ")?){1}([0-9]{2}(" ")?){3}
  ```

  Esta expresión es similar, solo que permite que los 6 últimos finales se agrupen en tres grupos de 2 cifras, en lugar de dos grupos de 3 cifras. De nuevo, se permite que haya un espacio entre los grupos de cifras y tras el prefijo internacional (si lo hay).

Hemos limitado a estos casos, puesto que son los más habituales en España. No obstante, se podrían añadir más expresiones regulares para detectar otros formatos de números de teléfono. Tenemos por tanto la siguiente regla, la cual fuerza a que se cumpla una de las dos expresiones regulares anteriores, y esto sea lo único que se encuentre en la línea:

```
^{tfno_espanol_grupos3}|{tfno_espanol_grupos2}$   {tfno_pais["España"].insert(yytext);}
```

Como vemos, los teléfonos detectados los almacenamos en un conjunto asociado a la clave `España`, de forma que sería muy fácilmente generalizable a un mayor número de países. Un ejemplo de funcionamiento de este programa se encuentra en la Figura 1.

3. Comprobación de Direcciones de Correo Electrónico

Este analizador se encuentra en el archivo `EMAIL/regex_email.l`. Para compilarlo, se debe ejecutar el siguiente comando:

```
$ flex++ -o regex_email.cpp regex_email.l
$ g++ -Wall -o regex_email regex_email.cpp -lfl
```

Para ejecutarlo, se debe ejecutar el siguiente comando:

```
$ ./regex_email <fichero>
```

donde `<fichero>` es el archivo que se desea analizar en busca de direcciones de correo electrónico.

De forma alternativa, se recomienda usar el `makefile` proporcionado usando la siguiente regla:

```
$ make email
```

En este caso, el archivo de texto que se analizará es `EMAIL/email.txt`, por lo que deberán almacenarse en ese archivo las cadenas que se deseen procesar.

Este programa analiza el archivo pasado en búsqueda de direcciones de correo electrónico, informando cuáles son válidas y cuáles no. Respecto a la validación de correos, en internet hay gran cantidad de expresiones regulares disponibles, la mayoria de una alta complejidad que usan al RFC 3696, que es la documentación en la cual se especifica qué formato han de tener las direcciones de correo electrónico. No obstante hemos optado por una expresión regular más sencilla hecha por nosotros, la cual detecta la gran mayoria de correos electrónicos válidos en la actualidad. La expresión regular que hemos usado es la siguiente:

```
[a-zA-Z0-9]+((\.|-)?[a-zA-Z0-9]+)*@[0-9]*[a-zA-Z]+[a-zA-Z0-9]*\.(com|es|org)
```

Las limitaciones que esta expresión regular establece son:

- Distinguiremos entre la parte local (antes del @) y dominio (tras el @).
- Respecto a la parte local, ha de ser una sucesión de caracteres alfanuméricos junto a guiones y a puntos, teniendo en cuenta que:
  - Ha de empezar y terminar por un caracter alfanumérico.
  - Tras cada guion o punto ha de ir un carácter alfanumérico, de forma que no haya dos símbolos especiales consecutivos.
- Respecto al dominio, antes del punto ha de haber una sucesión de caracteres alfanuméricos conteniendo al menos una letra, y tras el punto tan solo se admiten tres posibilidades: `com`, `es` u `org`.

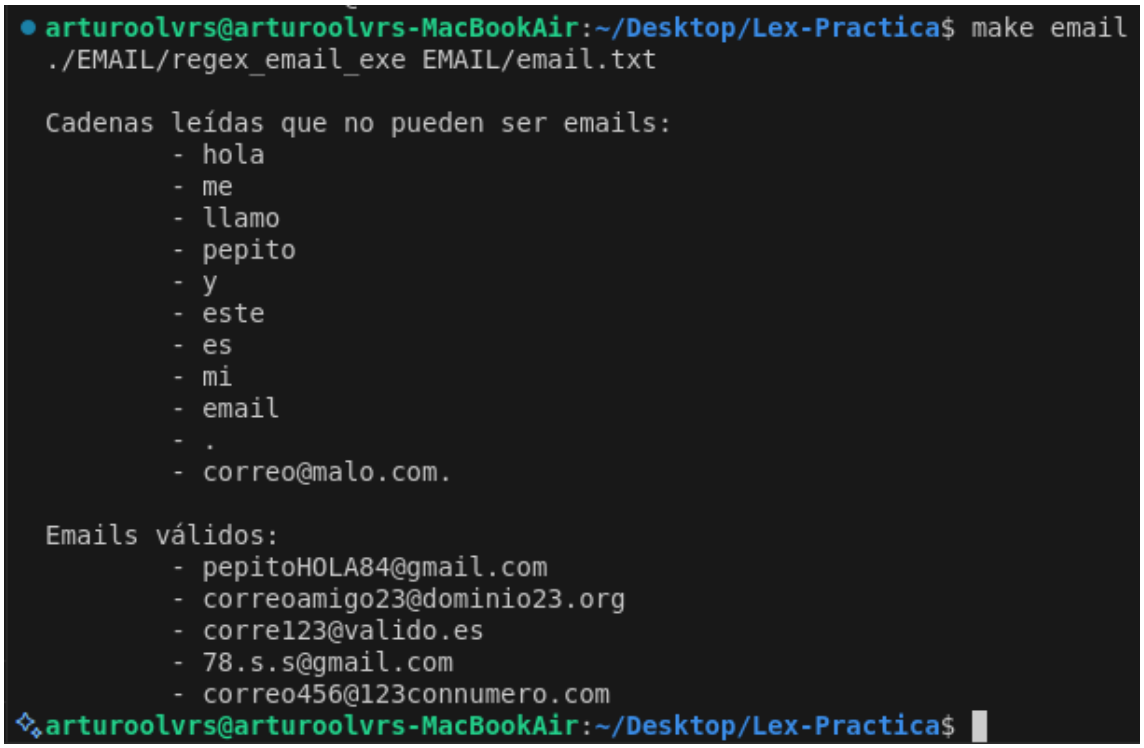

Figura 2: Ejemplo de ejecución del programa `regex_email`.

Somos conscientes de que esta expresión regular no es correcta (el correo electrónico institucional de la UGR no sería válido, por ejemplo), pero nos hemos limitado a ella para usar así una expresión regular legible. Un ejemplo de funcionamiento de este programa se encuentra en la Figura 2. Notemos que el correo `correo@malo.com.` no se ha dado como válido puesto que está seguido justo de un punto, por lo que lo detecta como parte del dominio.

4. Comprobación de DNIs y de NIEs

Este analizador se encuentra en el archivo `DNI-NIE/regex-dni-nie.l`. Para compilarlo, se debe ejecutar el siguiente comando:

```
$ flex++ -o regex-dni-nie.cpp regex-dni-nie.l
$ g++ -Wall -o regex-dni-nie regex-dni-nie.cpp -lfl
```

Para ejecutarlo, se debe ejecutar el siguiente comando:

```
$ ./regex-dni-nie <fichero>
```

donde `<fichero>` es el archivo que se desea analizar en busca de DNIs y NIEs.

De forma alternativa, se recomienda usar el `makefile` proporcionado usando la siguiente regla:

```
$ make dni-nie
```

En este caso, el archivo de texto que se analizará es `DNI-NIE/dni-nie.txt`, por lo que deberán almacenarse en ese archivo las cadenas que se deseen procesar.

Este programa analiza el archivo pasado en búsqueda de DNIs y NIEs, informando cuáles son válidos y cuáles no. La validación de DNIs y NIEs es más compleja, ya que por un lado se buscan aquellos candidatos a ser válidos (los que cumplan una expresión regular concreta, que explicaremos a continuación) y, por otro lado, ha de comprobarse que la letra del DNI o NIE sea la correcta. Respecto al patrón que han de seguir para poder ser válidos, hemos usado las siguientes expresiones regulares:

- Para los DNIs:

  ```
  [0-9]{8}(" "|"-")?[A-Z]
  ```

  Como el lector sabe, el DNI está formado por 8 dígitos seguidos de una letra en mayúsculas. Se permite que haya un espacio o un guión tras los 8 dígitos.

- Para los NIEs:

  ```
  [XYZ](" "|"-")?[0-9]{7}(" "|"-")?[A-Z]
  ```

  En este caso, el NIE puede empezar por una X, Y o Z, seguido de 7 dígitos y una letra en mayúsculas. De nuevo, se permite que haya un espacio o un guión antes y tras los dígitos.

Las reglas que hemos usado para detectar DNIs y NIEs son sencillas, y se muestran a continuación:

```
{DNI}       {dnis.push_back(yytext);}
{NIE}       {nies.push_back(yytext);}
```

De esta forma, se almacenan en dos vectores los DNIs y NIEs detectados, respectivamente. No obstante, llegados a este punto tenemos los que pueden ser válidos, pero no se ha probado su validez, puesto que falta comprobar el dígito de control. Esta comprobación se ha realizado siguiento la documentación oficial proporcionada por el Ministerio del Interior.

- Para la comprobación de DNIs, la letra se calcula como el resto de dividir el número del DNI entre 23, y según ese resto se elije una letra concreta proporcionada por la web previamente mencionada. El DNI será válido si la letra que se ha calculado es la misma que la que se ha introducido.

- Para la comprobación de NIEs, se sustituye la X, Y o Z iniciales por un 0, 1 o 2, respectivamente, y se sigue el mismo procedimiento que para los DNIs.

De esta forma, se puede comprobar si un DNI o NIE es válido o no. Un ejemplo de funcionamiento de este programa se encuentra en la Figura 3. Como vemos, aunque haya una gran cantidad de DNIs reconocidos como tal, la mayoría de ellos no son válidos tras comprobarlos con el dígito de control.

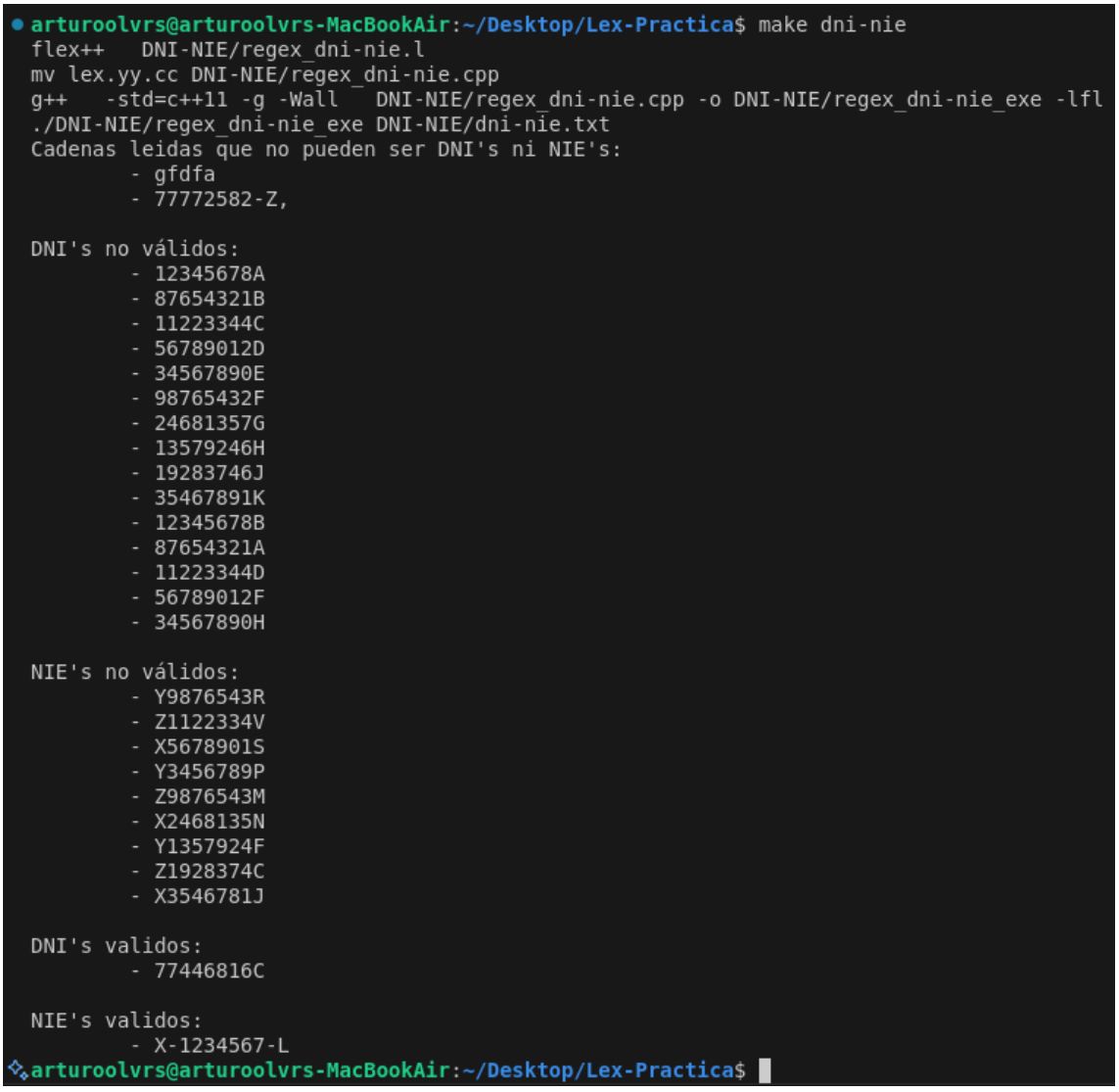

Figura 3: Ejemplo de ejecución del programa `regex-dni-nie`.

5. Comprobación de Cuentas Bancarias

Este analizador se encuentra en el archivo `Bancos/regex_bancos.l`. Para compilarlo, se debe ejecutar el siguiente comando:

```
$ flex++ -o regex_bancos.cpp regex_bancos.l
$ g++ -Wall -o regex_bancos regex_bancos.cpp -lfl
```

Para ejecutarlo, se debe ejecutar el siguiente comando:

```
$ ./regex_bancos <fichero>
```

donde `<fichero>` es el archivo que se desea analizar en busca de cuentas bancarias.

De forma alternativa, se recomienda usar el `makefile` proporcionado usando la siguiente regla:

```
$ make bancos
```

En este caso, el archivo de texto que se analizará es `Bancos/bancos.txt`, por lo que deberán almacenarse en ese archivo las cadenas que se deseen procesar.

Este programa analiza el archivo pasado en búsqueda de cuentas bancarias, informando finalmente la entidad bancaria a la que corresponde cada cuenta. En primer lugar, hemos de tener en cuenta que se restringe a las cuentas bancarias españolas, ya que es el único país cuyas cuentas bancarias conocemos. Tras detectar las cuentas bancarias mediante una expresión regular que más adelante se detallará, estas se validan también mediante dos dígitos de control de forma similar a los DNIs, y finalmente, de las cuentas bancarias válidas, informamos de su entidad bancaria.

La expresión regular que hemos usado para detectar cuentas bancarias españolas es la siguiente:

```
ES[0-9]{2}(" "[0-9]{4}){5}
```

En este caso, el IBAN de la cuenta bancaria ha de empezar por `ES` (por ser españolas), seguido de dos dígitos (serán los dígitos de control), y a continuación 5 grupos de 4 dígitos separados por un espacio, que componen el número de cuenta. De todas estas cuentas (que son las candidatas a ser válidas), se ha de comprobar que los dígitos de control sean correctos. Para ello, se ha usado la documentación oficial proporcionada por el Banco de España, que proporciona una serie de pasos a seguir para comprobar la validez de una cuenta bancaria. En resumen, se han de seguir los siguientes pasos:

1. Se toman los 20 dígitos que forman el número de cuenta (excluyendo las letras del país, “ES”, y los dos dígitos de control).

2. Tras ellos, para cada letra del código del país, se le asigna un número. A la A se le asigna el 10, a la B el 11, y así sucesivamente hasta la Z. En el caso de “ES”, se le asigna el 14 y el 28, y ambos se añaden al final de los 20 dígitos.

3. Por último, y como desconocemos los dos dígitos de control (queremos calcularlos), se añaden al final dos ceros.

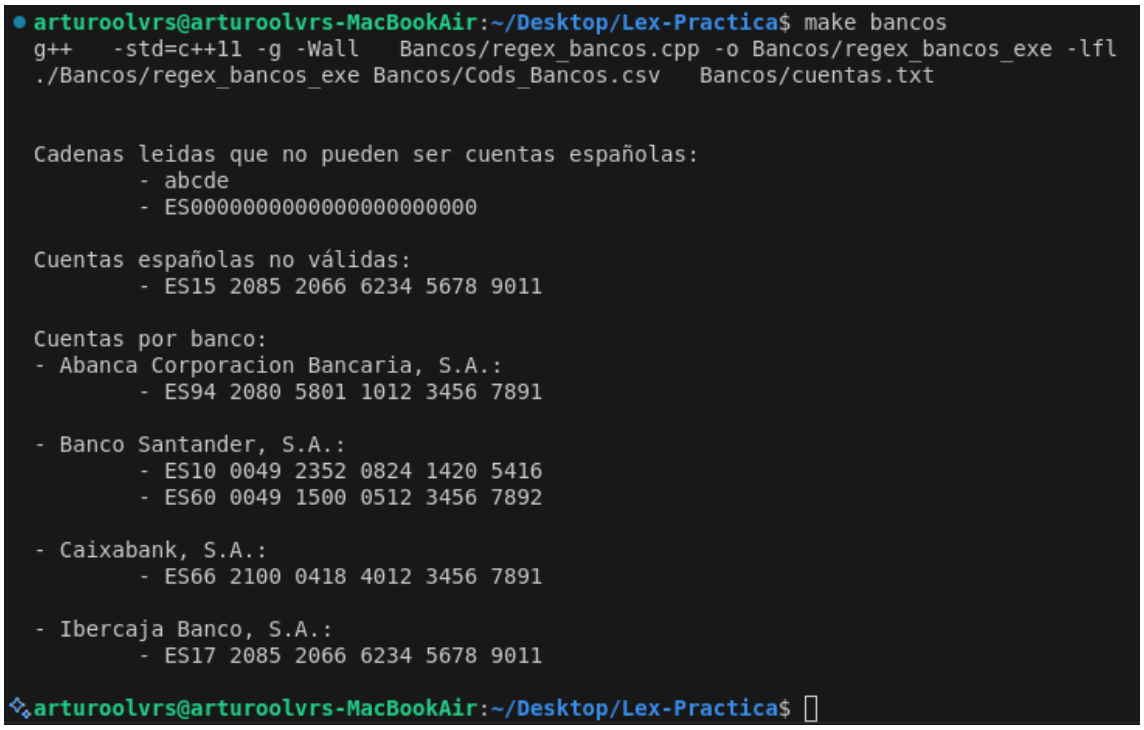

Figura 4: Ejemplo de ejecución del programa `regex_bancos`.

4. A este número obtenido (con 26 cifras[1]) se le calcula el resto de dividirlo entre 97.

5. A 98 se le resta el resto obtenido, y este resultado (que será de dos cifras (pudiendo ser el primero un 0)) será el número de control. Estos dos dígitos han de ser los mismos que los dos dígitos de control que se han introducido para que el IBAN sea válido.

Una vez validados los IBAN, buscamos informar de la entidad bancaria a la que pertenecen. Para ello, hemos usado también la información oficial del Banco de España, ya que los primeros 4 dígitos tras los dígitos de control corresponden a la entidad bancaria. En la documentación proporcionada por el Banco de España, se detalla qué entidad bancaria corresponde a cada número, y hemos usado esta información para informar al usuario de la entidad bancaria a la que pertenece la cuenta bancaria. Un ejemplo de funcionamiento de este programa se encuentra en la Figura 4.

6. Menú de Opciones

Llegados a este punto, en el que hemos explicado cada una de las distintas funcionalidades que hemos implementado, vamos a explicar otra forma de probarlas, que es mediante el menú que hemos creado. Este menú se encuentra en el archivo `menu.cpp`, y al no contener ningún comando de flex, se compila de la forma habitual:

[1]Este aspecto hay que tenerlo en cuenta a la hora de calcular el resto, puesto que no se puede almacenar entero en un tipo de dato numérico de `C++`.

```
$ g++ -Wall -o menu menu.cpp
```

Para ejecutarlo, se debe ejecutar el siguiente comando:

```
$ ./menu
```

De forma alternativa, se recomienda usar el `makefile` proporcionado usando la siguiente regla:

```
$ make menu
```

Al ejecutar este programa, se mostrará un menú con las distintas opciones que se pueden elegir, y se podrá seleccionar la que se desee. Las opciones que se pueden elegir son las siguientes:

1. Comprobación de números de teléfono.
2. Comprobación de direcciones de correo electrónico.
3. Comprobación de DNIs y NIEs.
4. Comprobación de cuentas bancarias.

Una vez elegida cualquiera de las opciones, se pide por la entrada estándar al usuario que introduzca el texto que desea analizar, y este se le pasará al programa correspondiente. Un ejemplo de funcionamiento de este menú se encuentra en la Figura 5.

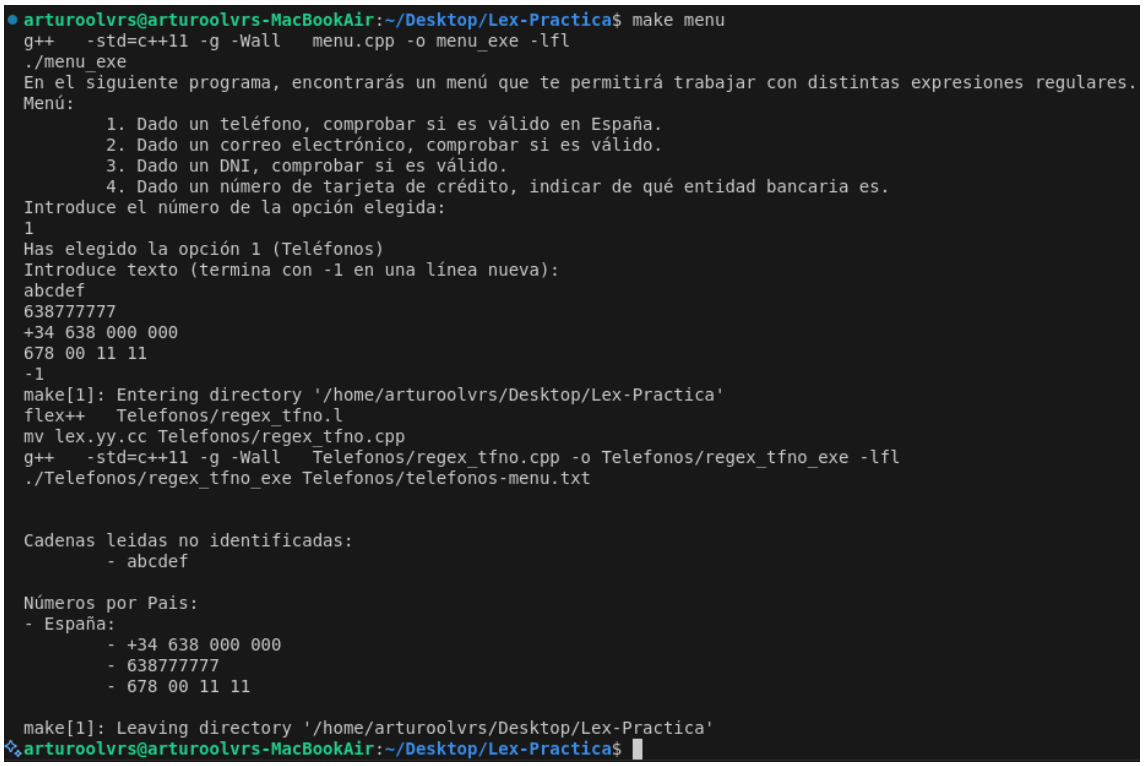

Figura 5: Ejemplo de ejecución del menú.