



Instituto Tecnológico de Costa Rica

Arquitectura de Computadores

Proyecto NASM 1: Expresiones Regulares

Profesor: Erick Hernández Bonilla.

Estudiantes: Kevin Lobo Chinchilla (2015088135)
Víctor Chaves Díaz (2015107095)
Jorge Gonzalez Rodriguez (2015083567)

Semestre II

2015

I. Introducción

El presente proyecto trata sobre la implementación de un tokenizer utilizando expresiones regulares que se definen como una serie de caracteres que forman un patrón de tal manera que se puede comparar dicho patrón con una cantidad grande de caracteres para encontrar coincidencias.

Según wikipedia, en el área de la programación las expresiones regulares son un método por medio del cual se pueden realizar búsquedas dentro de cadenas de caracteres. Sin importar si la búsqueda requerida es de dos caracteres en una cadena de 10 o si es necesario encontrar todas las apariciones de un patrón definido de caracteres en un archivo de millones de caracteres, las expresiones regulares proporcionan una solución para el problema. Adicionalmente, un uso derivado de la búsqueda de patrones es la validación de un formato específico en una cadena de caracteres dada, como por ejemplo fechas o identificadores.

II. Explicación del diseño y Arquitectura

El presente proyecto fue programado en ensamblador con sintaxis NASM, para el sistema operativo Linux. Se diseñó de tal manera que se dividiera en 3 grandes partes:

- Obtención y separación de entradas: Como su nombre lo dice, la primera parte consiste en obtener la entrada ingresada por medio de un archivo de texto y separarla en 3 partes:
 - Regex
 - Texto por reemplazar
 - Texto donde buscar

Esta separación se dá de manera lógica, utilizando diferentes registros como punteros al inicio de cada sección. En el caso del texto en donde buscar, se utiliza otro registro como puntero a su final. Se debe tomar en cuenta que en caso de que el editor de texto no agregue un salto de línea al final del texto, el programa se encarga de hacerlo. Esto debido al algoritmo que se utilizó para determinar cuándo se ha llegado al final del texto donde buscar hace uso del salto de línea como indicador.

- Chequeo de matches: Una vez separadas las entradas, se procede a hacer el chequeo adecuado de matches. En este apartado, se comprueba la aparición de caracteres especiales, como lo son el punto, los sets, entre otros que son detallados en la siguiente sección.
- Impresión del texto con reemplazos por consola: Para efectos de este proyecto, los paréntesis en las expresiones regulares connotan la sección del texto encontrado con las que se debe hacer reemplazo en el nuevo texto. En dicho texto, la parte que se sustituye será señalada mediante la simbología de \$#, donde # puede ir de 1 a 8.

Se utiliza un buffer de entrada de 4256 bytes ya que ese será el tamaño máximo de la entrada. 80 bytes para la regex, 80 bytes para el texto donde reemplazar y 4K para el texto donde buscar.

Cabe resaltar que se hace uso de otros buffers de tamaño 4K con el fin de evitar errores en los que la memoria reservada sea insuficiente.

Algunos buffers fueron creados debido a que los registros disponibles fueron usados en su totalidad.

El programa no acepta entradas con el texto a buscar completamente vacío. Para indicar que no hay texto a buscar, se deben agregar 3 saltos de línea.

III. Descripción Técnica

3.1 Estructuras:

Se utilizaron buffers para almacenar las diferentes entradas y salidas del programa.

- inputBuffer: Reserva 4256 bytes utilizados para almacenar la entrada principal del programa.
- replacedTextBuffer: Almacena el string por imprimir en pantalla.
- dollar#: Se usaron 8 buffers de este tipo. En ellos se almacenan las palabras que hacen match y se encontraban encerradas entre paréntesis en la expresión regular de la entrada.

- frontTrackSuccess: Buffer de un byte utilizado dentro de frontTracking (método para realizar front tracking en ciertas expresiones) para indicar éxito o fracaso del proceso ya mencionado
- nextCharDirection: Buffer de 8 bytes utilizado para indicarle a los procesos que hagan uso del front tracking de en donde en el texto deberían de parar de hacer match.
- boolCrossMadeMatch: Buffer de un byte utilizado en los procesos que hagan uso del multiplicador +, y sirve para indicarle al proceso si ya previamente se había hecho match, esto con la finalidad de asegurar que el multiplicador cruz haga match una o más veces.

3.2 Algoritmos:

Obtención de la entrada y división de la misma:

Para la obtención de la entrada se utilizará el típico algoritmo de entrada:

```

mov rax, 0                ; sys_write
mov rdi, 0                ; stdin
mov rsi, inputBuffer      ; Donde almacenar lo leído
mov rdx, 4256             ; Cantidad de bytes por leer
syscall                   ; Llamada al sistema

```

La entrada anterior consta de 3 partes distintas:

1. Expresión Regular
2. Nueva expresión por crear
3. Texto en el que se debe buscar la expresión

Debido a esto, es necesario dividir de alguna manera las 3 partes con el fin de trabajar de manera independiente sobre cada una de ellas. Para llevar esto a cabo se realiza lo siguiente:

Se utiliza el registro R10 como puntero al inicio de la entrada, es decir, al inicio de la expresión regular. Posteriormente, con ayuda de las instrucciones repne y scasb se busca en la entrada el primer salto de línea. Al encontrarlo y apoyándose en el hecho de que scasb deja rdi en el caracter siguiente al encontrado, se utiliza el registro R12 como puntero al inicio de la nueva expresión por crear y de la misma manera se busca otro salto de línea.

R14 se usa como puntero al inicio del texto en el cual buscar la expresión regular, y haciendo uso de la cantidad de caracteres leídos almacenados en rax por la llamada a syscall, se puede determinar donde termina dicho texto y al final de este se deja el R15 con el fin de tener una condición de salida del programa.

Se debe tomar en cuenta que según el editor de texto utilizado para crear la entrada al programa, el último caracter puede ser o no un salto de línea, por lo que se decide agregar uno en caso de que el editor de texto no lo haga. Esto con el fin de facilitar la creación de próximos algoritmos.

Chequeo de Matches:

Para el chequeo de matches, la funcionalidad principal del programa, se utiliza un procedimiento llamado "matchCheck" el cual está separado en diversos procedimientos locales que se describirán posteriormente.

Justo después de la separación de la entrada, se realiza una llamada a esta función que es la encargada de verificar si alguno de los caracteres, tanto normales como especiales, de la expresión regular coincide con el caracter actual del texto en el cual buscar.

La comparación y avance en los textos se realiza de la siguiente manera: Como se mencionó anteriormente, R10 y R14 apuntan a la regex y al texto donde buscar respectivamente. Los registros "al" y "cl" almacenan el caracter al que apunta R10 y R14 respectivamente y luego se comparan ambos registros con el fin de buscar coincidencias.

Dado que algunos caracteres especiales aparecen después del caracter sobre el que actúan, el R11 se sitúa siempre una posición después del R10 para saber cómo proceder con el caracter actual en caso de que se encuentre antes de un caracter especial. En el registro "bl" se almacena el caracter al que apunta R11, por lo que

antes de comparar “al” con “cl” se compara ese registro con los diferentes caracteres especiales para determinar cómo proseguir. Cabe resaltar que el RBX se utiliza como booleano para determinar si hubo match o no al retornar de un procedimiento de caracteres especiales.

Match con caracteres especiales:

- **Grupos:** Los grupos (o sets) corresponden a un conjunto de caracteres en medio de corchetes, de los cuales uno puede hacer match con el texto. Su implementación se dá en el procedimiento setMatch, en donde se decide si se debe ejecutar la variante normal (aquella en donde uno de los caracteres del conjunto debe hacer match con el texto) o la variante negada (el texto no debe hacer match con ninguno de los caracteres listados, y se indica con un ^ al principio del set). Para cada caso existe un procedimiento, nombrados setMatchChars y setNotMatchChars para la variante normal y la variante negada, respectivamente, pero en esencia ambas realizan el mismo trabajo: se empieza a recorrer desde el primer caracter luego del corchete de apertura (en el caso de la variante negada, desde el ^) hasta el corchete de cierre, revisando si cada uno de los caracteres en medio hace match, en cuyo caso se indica, por medio del registro rbx, éxito (o fracaso en el caso de la variante negada). Ambos procedimientos salen cuando encuentran el corchete de cierre, aprovechando la implementación de ciclo para revisar los caracteres para poder dejar el puntero del regex listo para la siguiente instrucción.
- **Caracter “ . ”:** Durante el flujo de “matchCheck”, el registro “al” contiene el caracter al que apunta R10. En caso de que este sea un “ . ”, se salta a un procedimiento local llamado “.dotMatch” que junto a otros procedimientos requeridos para casos especiales, hace match con cualquier caracter que encuentre.
- **Caracter “ ^ ”:** Al detectar este caracter con el R10, se procede a llamar a una función local “.startLine” la cual a su vez llama al procedimiento startLine, este comprueba que el caracter anterior a R14 sea un salto de línea, si lo es, se coloca un 1 en RBX de lo contrario un 0.

- **Caracter “ \$ “:** Cuando se encuentra este caracter en la expresión regular, se procede al procedimiento local “.endLine” el cual compara R14 con un salto de línea o un nulo, que son los dos posibles valores que podría tener R14 para que el último caracter con el que se logró hacer match se encuentre antes del final de línea. De ser así, el programa salta a “.match”, de lo contrario salta a “.noMatch”.
- **Caracter “ ? ” (Cero o Uno):** Para trabajar con este caracter se utiliza el R11 para detectar su aparición. En caso de ser detectado se procede a “.questionMatch” y este llama al procedimiento “questionMatch” que se encarga de comparar “al” (caracter por revisar si se encuentra o no) con “cl” (caracter actual del texto donde buscar). En caso de que el caracter aparezca, coloca un 1 en RBX para que se llame a “.match”, si no aparece no se llama a este procedimiento pero no afecta a la búsqueda de match ya que el “?” permite que el caracter no se encuentre. En caso de que el “?” se encuentre después de un grupo, se realiza el mismo proceso que con un caracter solo pero primero se comprueba si el grupo hace match con el caracter actual del texto donde se busca.
- **Caracter “+” (Uno o más):** Al igual que con el “?”, este caracter es detectado gracias a la utilización del R11. Si se detecta se puede saltar a dos posibles procedimientos: “.firstCrossMatch” y “.crossMatch”. Esto se realizó de esta manera ya que la primera vez que se deben tomar acciones al encontrar un “+” se almacena la posición actual de R14 en R13, esto con el fin de determinar si en algún momento hubo match o si el caracter apareció. Otro objetivo del R13 en este caso es mantener la posición inicial de R14 en caso de que no haya match ya que simplemente significa que el caracter en cuestión no apareció y se debe saltar a .noMatch sin modificar la posición de R14.
- **Caracter “ * ” (Cero o más):** Al igual que los caracteres especiales anteriores, su presencia se detecta mediante el R11. Cuando esto sucede, se lleva a cabo un algoritmo similar al de “ + “, la primera vez que se detecta se procede a “.firstAsteriskMatch” y las siguientes apariciones del caracter en cuestión se accede a “.asteriskMatch”. De igual manera, R13 se deja

apuntando a la posición “inicial” de R14 al iniciar matches de un caracter con “ * “. Sólo que en este caso, si se determina que R14 y R13 son iguales y que por lo tanto no se encontró el caracter buscado se toma como si nada hubiera pasado, no se procede a “.match” ni a “.noMatch” ya que el asterisco permite la no aparición del caracter. Por lo tanto, simplemente se aumentan los registros R10 y R11 para que apunten a los caracteres siguientes del “ * “.

- **BackSlash “ \ “:** La funcionalidad de este caracter consiste en hacer a los caracteres especiales anteriores trabajar como si fueran caracteres literales, es decir, permite buscarlos en el texto como cualquier otro caracter. La forma en que se implementó es simple, consiste en determinar si R10 apunta a este caracter, de ser así se salta a “.backSlashFound” que incrementa R10 y R11 para intentar hacer match con el siguiente caracter que se supone es uno de los especiales y se compara inmediatamente con lo que apunta R14.

Nota: Los caracteres especiales “ ? ”, “ + “ y “ * “ requirieron de un algoritmo especial de Front Tracking que se explicará más adelante.

Paréntesis, dólar y texto de reemplazo:

Para efectos de este proyecto, los paréntesis de la expresión regular serán utilizados para delimitar el texto que se quiere mostrar una vez la regex (regular expression) ha hecho match completamente. Por otro lado, el texto de reemplazo es aquel que contendrá en su cuerpo “ \$# ” donde # es un número del 1 al 8. Esto significa que aquella primera sección de la regex que se encuentre entre paréntesis sustituirá a \$1 en el texto por reemplazar.

Para lograr esto, se utilizaron 8 buffers, uno para cada dólar, los cuales se van llenando de acuerdo al match dentro del paréntesis indicado. Para saber cual buffer se debe ir llenando, se utiliza el registro RDI como contador, el cual aumenta por cada “)” que encuentre, así el próximo “(“ que encuentre sabrá cuál buffer llenar.

El algoritmo para la implementación de los paréntesis es simple, si R10 apunta a un “ (“ este se ignora pero se pone RSI en 1 y continúa con el flujo normal del programa. Si encuentra un “ (“ pone RSI en 0 e incrementa RDI.

Para el manejo de los buffer dólar se debe conocer la longitud del string que se va almacenando para saber en qué parte del buffer almacenar el caracter actual. Dado

que se utilizaron los registros disponibles para el resto de algoritmos, en este caso se usó un buffer que almacena las direcciones en el stack donde se mantiene la longitud de cada buffer dólar.

Cabe resaltar que el registro RSI es utilizado como bandera, para determinar si se encuentra buscando match con caracteres de la regex entre paréntesis, esto para saber si llenar el buffer adecuado. Debido a lo anterior, se utiliza un procedimiento externo llamado "callDBS" que decide si llamar al procedimiento "dollarBufferSelection" de acuerdo al estado del RSI.

Ahora bien, este procedimiento "dollarBufferSelection" fue implementado con una estructura de control switch pero programada en ensamblador, en la cual los casos son determinados por el registro RDI, que gracias a este se sabe cual buffer llenar.

En el caso de que la expresión regular no haga match con alguno de los caracteres de R14, los buffer de dólar deben limpiarse para evitar errores en futuros llenados.

Para esto se utilizan los procedimientos "clearDollarBuffers1" y "clearDollarBuffers2" el primero de ellos verifica si RDI es igual a 0, si lo es deja de llamar a "clearDollarBuffers2", de lo contrario lo llama y decrementa a RDI, esto para asegurar la limpieza de cada uno de los buffers. "clearDollarBuffers2" al igual que "dollarBufferSelection" utiliza una estructura similar a switch para seleccionar el buffer por limpiar. La limpieza consiste en poner en cada uno de los bytes del buffer un caracter null.

En cuanto al texto de reemplazo, el algoritmo para la sustitución de "\$#" consiste en ir llenando el buffer "replacedTextBuffer" con los caracteres del texto donde reemplazar, en este caso, R12 apunta al inicio de ese texto y moviendo R12 es como se va almacenando caracter por caracter en el buffer mencionado anteriormente. Cuando R12 encuentra un "\$" se procede a "addDollarBuffer" y de acuerdo al número seguido de \$ se agrega el buffer dólar correspondiente al buffer "replacedTextBuffer". Antes de iniciar el agregado del buffer dólar se corrobora su longitud, si es 0 simplemente se agrega el "\$#" al texto de salida. Lo mismo sucede si en el texto viene un "\$#" con $\# \leq 0$ ó $\# \geq 9$.

Front Tracking:

El front tracking se define como una funcionalidad que utilizan todos los multiplicadores (?, * y +) para definir en donde deberían de finalizar la labor de multiplicidad. Un ejemplo, si se tiene la expresión `.+c` y el texto `abc`, entra en función el front tracking, indicando a la expresión `.+` que cuando encuentre en el texto la `c`, debe dejar de hacer match y dar paso al siguiente caracter a hacer match.

Esta funcionalidad se implementa en el proyecto por medio del procedimiento `frontTracking`, el cual utiliza los registros `r11` para moverse a través del regex y el `r14` para moverse a través del texto. Este procedimiento se encarga de indicarle a quien lo haya llamado, por medio de `frontTrackSuccess`, si se puede hacer match o no. En caso de que se pueda hacer match, se indica en `nextCharDirection` la dirección en memoria del caracter en el texto en donde se debe dejar de hacer match y dar paso a la siguiente instrucción.

Todos los posibles casos que llamen a este procedimiento empiezan por hacer una serie de pushes de registros que se utilizan durante el transcurso de la vida del front tracking, y se procede a llamar a `checkForChar`. Este subprocedimiento se encarga de revisar que tipo de patrón se encuentra luego del multiplicador, los cuales pueden ser un caracter cualquiera (inclusive puede ser un caracter acompañado de un multiplicador), un set o un caracter escapado. Caracteres como los parentesis son ignorados, procediendo a mover el registro `r11` al caracter siguiente del regex, y en el caso de que haya un dolar (final de linea), se procede a revisar hasta el final de la linea en el texto.

En el caso de que un caracter o un set vengan acompañados de un multiplicador, solo se revisa si este operador es un signo de pregunta (puede estar 0 o 1 vez) o un asterisco (0 o mas veces), pues estos dos casos son los unicos en los que puede no existir el caracter siguiente, por lo que se procede a revisar si en el texto existe coincidencia del caracter multiplicado, y en caso de que no exista, se procede a revisar con el siguiente caracter en el regex.

IV. Manual de Usuario

El programa consiste en la búsqueda de un patrón de texto en un texto, cuya salida en caso de que el patrón sea encontrado en el texto donde buscar será un nuevo texto previamente ingresado con partes especiales sustituidas.

Ejemplo simple de entrada:

H(ola)	<----- Patrón
H\$1, como estas?	<----- Nuevo texto.
Hola mi nombre es Enrique.	<----- Texto en el cual se debe buscar el patrón.

A continuación se explican detalladamente cada una de las partes de la entrada:

Patrón:

Conocido como expresión regular, será un pedazo de texto que debe buscarse.

Puede contener caracteres especiales:

- “ . “: Representa cualquier caracter. Por ejemplo, “H.la” como patrón encontrará cualquier palabra que inicie con H, tenga cualquier caracter después de esta y termine con la: Hola, Hala, Hfla, Hula, etc.
- “^”: Significa que cualquier caracter después de él debe estar obligatoriamente al inicio de una línea.
- “\$”: Denota que cualquier caracter antes de él debe estar obligatoriamente al final de una línea.
- **Grupos “[]”**: Denota que cualquier caracter que se encuentre dentro de paréntesis cuadrados debe coincidir con el caracter actual del texto donde se busca.
- “ ? “: Significa que el caracter antes de él puede aparecer cero o una vez en el texto.
- “ * “: Significa que el caracter antes de él puede aparecer cero o más veces.
- “ + “: Significa que el caracter anterior a él debe aparecer una o más veces.

- “ \ “: Se utiliza cuando se desea buscar un caracter especial en el texto donde buscar de manera literal. Es decir, “ \+” busca un “ + ” en el texto y no “ \ “ una o más veces.

Nota: La longitud del patrón no puede exceder los 80 caracteres.

Nuevo Texto:

En algunas ocasiones, si lo desea, el patrón puede venir con ciertas partes entre “()” esto significa que cuando el patrón se encuentre en el texto donde buscar, aquellas partes encerradas entre paréntesis sustituirán a los símbolos \$# en el texto nuevo. Donde $0 < \# < 9$.

Por ejemplo:

H(ola)

H\$1, como estas? <---- Nuevo texto

Hola mi nombre es Enrique

La salida del programa será “Hola, como estas?” ya que el patrón “H(ola)” estaba en el texto donde buscar y el \$1 en el texto nuevo se sustituyó por “ola” lo que se encontraba en el paréntesis.

Nota: La longitud del nuevo texto no debe exceder los 80 caracteres. Esto no significa que al realizar las sustituciones respectivas no pueda exceder dicho límite.

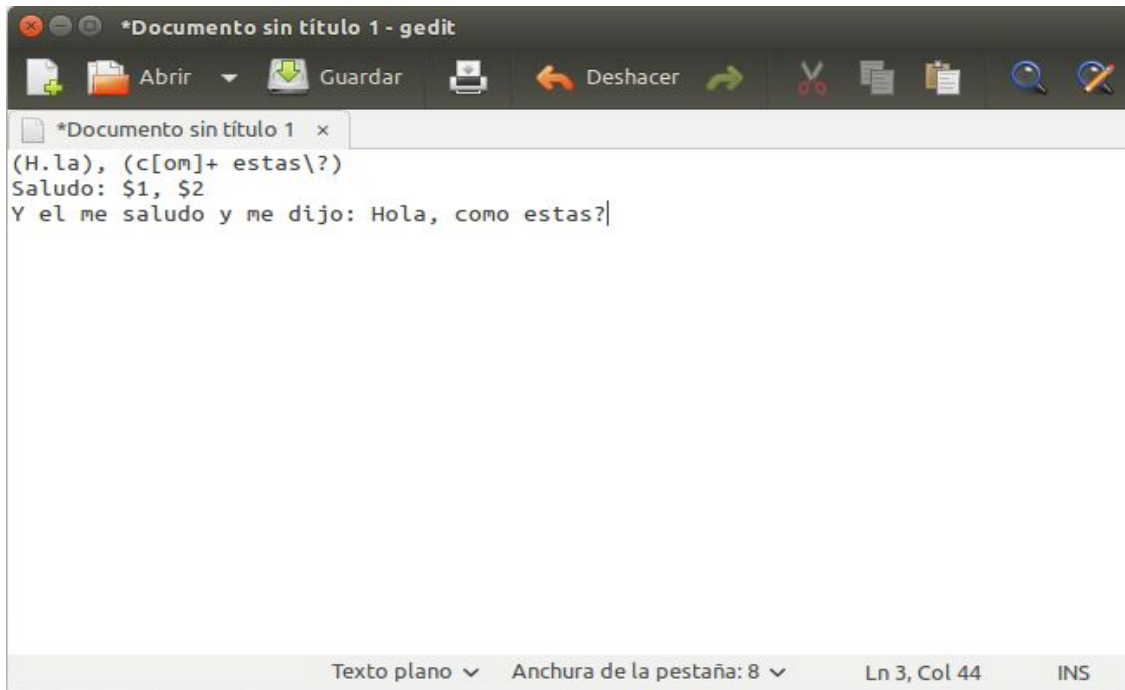
Texto donde buscar:

Texto cualquiera no mayor a 4096 caracteres de longitud. En él se buscará el patrón ingresado.

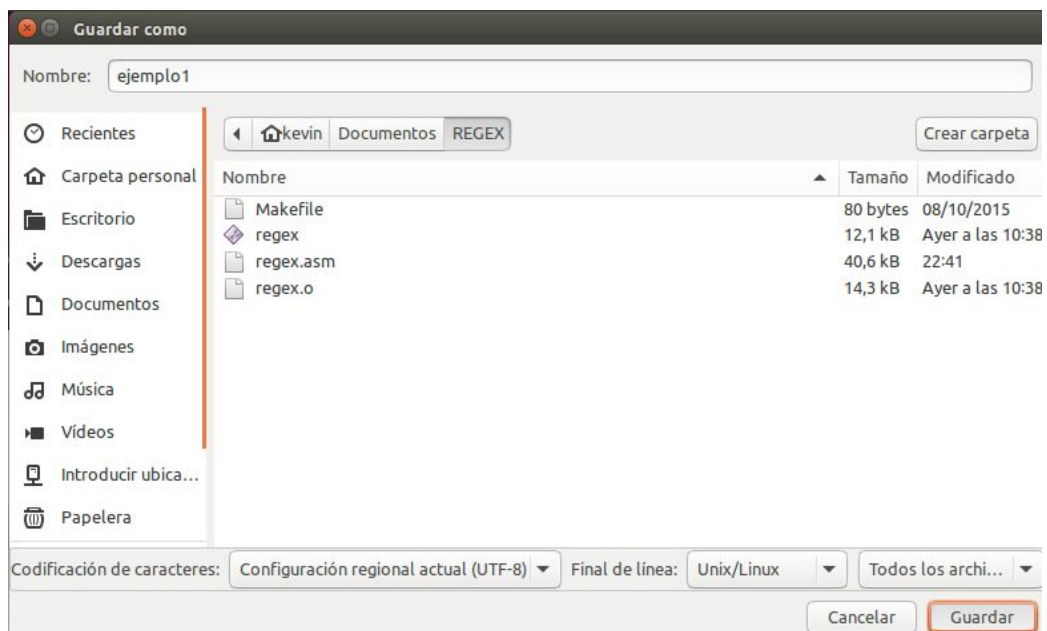
Instrucciones de uso:

Antes que nada es importante crear un archivo de texto que será la entrada principal del programa. Para esto puede utilizar su editor de texto favorito, en este caso se utilizará gedit.

1 - Escriba la entrada que desea para el programa. Observe el siguiente ejemplo:

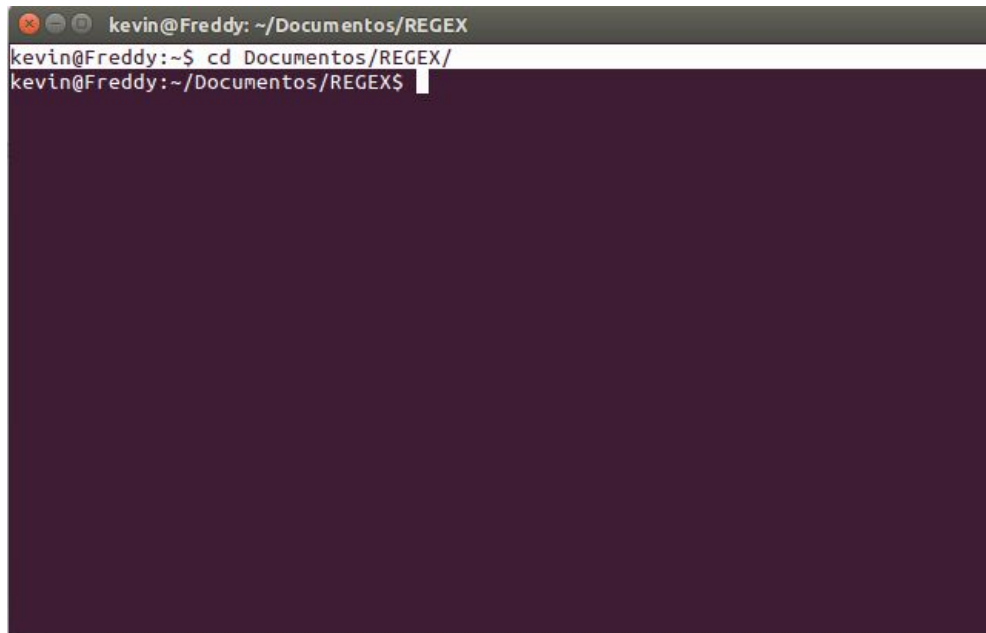


2- Guarde el archivo de texto dentro de la carpeta que contiene el ejecutable del programa.



3- Una vez almacenado el archivo, proceda a abrir la terminal de linux. Para esto, puede utilizar el atajo de teclado Ctrl+Alt+T.

4- Con la terminal abierta, diríjase a la carpeta en la cual se encuentra el ejecutable del programa. En caso de no poseer el ejecutable, debe asegurarse de tener en una carpeta un archivo llamado makefile y el archivo *.asm del programa. Para esto puede hacer uso del comando “cd” de la terminal. Observe el texto seleccionado en la consola:



```
kevin@Freddy: ~/Documentos/REGEX
kevin@Freddy:~$ cd Documentos/REGEX/
kevin@Freddy:~/Documentos/REGEX$
```

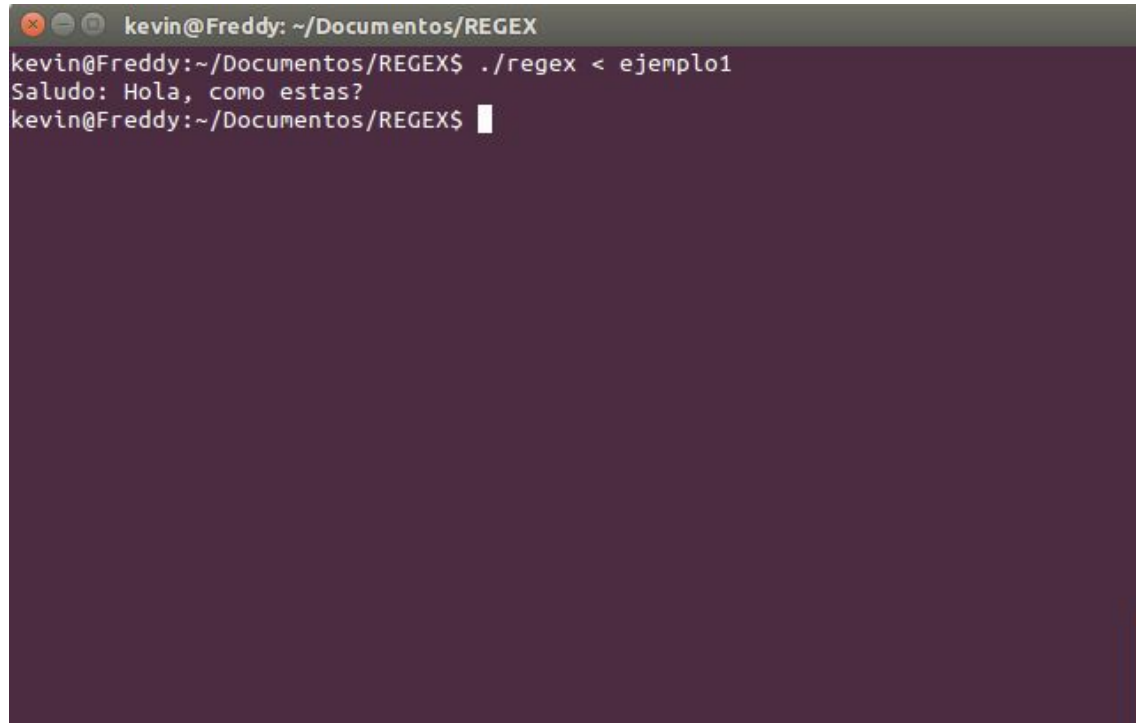
5- Una vez dentro de la carpeta, ingrese el comando “make” sin comillas, para crear el ejecutable en caso de que no lo tenga.

6- Una vez dentro de la carpeta y el ejecutable en ella, debe ingresar el siguiente formato de comando: “./ejecutable < archivoEntrada”. En nuestro ejemplo será: “./regex < ejemplo1” Sin comillas. Por ejemplo:



```
kevin@Freddy: ~/Documentos/REGEX
kevin@Freddy:~/Documentos/REGEX$ ./regex < ejemplo1
```

7- Presione la tecla ENTER. Verá la salida del programa con el nuevo texto y los símbolos \$# sustituidos por lo deseado:

A terminal window with a dark purple background and a light grey title bar. The title bar contains the text 'kevin@Freddy: ~/Documentos/REGEX'. The terminal shows the command './regex < ejemplo1' being executed, followed by the output 'Saludo: Hola, como estas?'. The prompt 'kevin@Freddy:~/Documentos/REGEX\$' is visible at the bottom of the terminal, with a white cursor character at the end.

```
kevin@Freddy: ~/Documentos/REGEX
kevin@Freddy:~/Documentos/REGEX$ ./regex < ejemplo1
Saludo: Hola, como estas?
kevin@Freddy:~/Documentos/REGEX$
```