Problemas introductorios sobre la shell Bash (Ejemplos resueltos)

September 15, 2023

Problema 1. Teniendo en cuenta que:

- El comando básico echo tiene la siguiente ayuda en la utilidad man:

```
NAME
    echo - Write arguments to the standard output.

SYNOPSIS
    echo [-neE] [arg ...]

DESCRIPTION
    Write arguments to the standard output.

    Display the ARGs, separated by a single space character and followed by a
    newline, on the standard output.

    Options:
      -n do not append a newline
      -e enable interpretation of the following backslash escapes
      -E explicitly suppress interpretation of backslash escapes

    ‘echo’ interprets the following backslash-escaped characters:
      \a alert (bell)
      \b backspace
      \c suppress further output
      \e escape character
      \E escape character
      \f form feed
      \n new line
      \r carriage return
      \t horizontal tab
      \v vertical tab
      \\ backslash
      \0nnn the character whose ASCII code is NNN (octal).  NNN can be
     0 to 3 octal digits
      \xHH the eight-bit character whose value is HH (hexadecimal).  HH
     can be one or two hex digits
      \uHHHH the Unicode character whose value is the hexadecimal value HHHH.
     HHHH can be one to four hex digits.
      \UHHHHHHHH the Unicode character whose value is the hexadecimal value
     HHHHHHHH. HHHHHHHH can be one to eight hex digits.

    Exit Status:
```

```
Returns success unless a write error occurs.
```

- El carácter Escape en bash es "\". En bash, el carácter Escape preserva el valor literal del siguiente carácter con la excepción de newline, dado que \newline es considerado continuación de línea.
- Los metacaracteres de bash son *space*, *tab*, *newline*, '|', '&', ';', '(', ')', '<' y '>'.

¿ Cuál será la salida en la terminal de los siguientes comandos? Razona en cada caso el resultado obtenido.

1. echo hola, mundo
2. echo hola, mundo\
3. echo -n hola, mundo
4. echo hola, mundo hola, mundo
5. echo hola, mundo\n hola, mundo
6. echo -e hola, mundo\n hola, mundo
7. echo hola, mundo\\n hola, mundo
8. echo -e hola, mundo\\n hola, mundo
9. echo -e hola, mundo\\nhola, mundo
10. echo -e "hola, mundo\nhola, mundo"
11. echo -e 'hola, mundo\nhola, mundo'
12. echo (hola, mundo)
13. echo \(hola, mundo\)
14. echo hola; mundo
15. echo hola| mundo
16. echo hola& mundo
17. echo hola\& mundo
18. echo hola * mundo
19. echo hola * mundo
20. echo hola > mundo
21. echo < mundo

Solución. En la solución de este ejercicio, para cada apartado se muestra el comando, la salida que realmente produce, y la explicación de la misma.

1.
   - echo hola, mundo
   - ```
     hola, mundo
     user1@host:~$
     ```
   - El comando echo muestra por la salida de la terminal (es la salida estándar), los argumentos. Por defecto añade un newline como último carácter.
     ```

2.
   - echo hola, mundo\
   - >
   - En este caso se termina la línea con el carácter Escape. Esto lo cambia todo por lo siguiente:
     Lo primero que hace bash en una línea de comando es un proceso que se llama Quoting o Entrecomillado. El Quoting se utiliza para que aquellos caracteres que tienen un significado especial en Bash, como los metacaracteres, lo pierdan y vuelvan a tener su significado literal. Entonces:
     - El carácter Escape \ es un Quoting que se aplica al siguiente carácter.
     - Al leer la línea de comandos, bash se encuentra con \seguido del metacaracter newline y por tanto según lo que se ha explicado, aplicaría el Quoting sobre newline. Si esto fuera así newline, tendría que tener su valor literal que es newline y se produciría un salto de línea. Esto no ocurre aquí, porque newline es precisamente una excepción para el carácter \. En lugar de devolverle su valor literal, lo que ocurre aquí es que lo convierte en una **continuación de línea**.
     - Bash recoge esto como un caso especial de Quoting cuando está en modo interactivo abriendo el prompt secundario para continuar con la línea.
     - Es importante señalar que después de aplicar el Quoting, bash elimina los indicadores de Quoting, en este caso \

3.
   - echo -n hola, mundo
   - `hola, mundouser1@host:~$`
   - La diferencia con el primer caso es la opción -n, que elimina en la salida de echo el último newline. Por eso aparece la salida de echo y a continuación bash escribe el prompt.

4.
   - echo hola, mundo hola, mundo
   - `hola, mundo hola, mundo`
     `user1@host:~$`
   - Los dos hola, mundo aparecen en la misma línea porque no se ha indicado un salto de línea. El prompt aparece en una línea separada porque echo introduce al final un carácter newline.

5.
   - echo hola, mundo\n hola, mundo
   - `hola, mundon hola, mundo`
     `user1@host:~$`
   - Probablemente, con la intención de separar los dos hola, mundo, cada uno en su línea, se introdujo el carácter de control permitido por echo \n (saltar a la siguiente línea). Sin embargo, el resultado no fue el esperado. Simplemente se añadió el carácter n después de mundo. Hay varias razones para que esto ocurra. La principal se debe al funcionamiento de bash. **Lo primero que hace bash en una línea de comando es un proceso que se llama Quoting o Entrecomillado. El Quoting se utiliza para que aquellos carácteres que tienen un significado especial en Bash, como los metacaracteres, lo pierdan y vuelvan a tener su significado literal**. Entonces:
     - El carácter Escape \ es un Quoting que se aplica al siguiente carácter.
     - Al leer la línea de comandos, bash se encuentra con \n y por tanto aplica el Quoting sobre "n". Este carácter no es especial, así que realmente nada cambia.
     - Después de aplicar el Quoting, bash elimina los indicadores de Quoting, en este caso \

     Por eso, el token resultante que le queda al comando es "mundon". La segunda razón que impediría la aplicación de la secuencia Escape de salto de línea es que como se puede comprobar en la ayuda, echo solo reconoce estas secuencias si se aplica la opción -e. Sin embargo, esta no es la razón que está produciendo el resultado mostrado, ya que en realidad el argumento que llega a echo ya no tiene dicha secuencia Escape como se ha explicado.

6.
   - echo -e hola, mundo\n hola, mundo
   - `hola, mundon hola, mundo`
   - Sin embargo, con la solución propuesta, tampoco se consigue el resultado esperado. Esto se debe a que el intérprete bash actúa siempre antes de que se ejecute el comando. En el caso del ESCAPE lo trata de aplicar, y esto consiste en sustituir la secuencia "\n", por el car'acter literal que le sigue, en este caso "n". Que es lo que se muestra como resultado en la terminal.

7.
   - echo hola, mundo\\n hola, mundo
   - `hola, mundo\n hola, mundo`
     `user1@host:~$`
   - En este caso, se ha utilizado Escape para deshabilitar el siguiente carácter Escape durante la actuación de bash. De esta manera, el resultado es que tras el procesamiento de bash, el segundo argumento de echo en este caso es: mundo\nhola, mundo. Sin embargo tampoco se consigue el objetivo de separar las dos líneas, porque echo ha sido llamado con las opciones por defecto, donde no se interpretan las secuencias Escape, y no se logra el salto de línea deseado.

8.
   - echo -e hola, mundo\\n hola, mundo
   - `hola, mundo`
     ` hola, mundo`
     `user1@host:~$`
   - Ahora sí se produce el salto de línea, ya que hemos incorporado la opción -e para que el comando echo interprete las secuencias Escape. Existen otras formas de desactivar la acción de Escape en bash, básicamente mediante el uso del entrecomillado. Se observa un efecto inesperado, la aparición de un espacio delante del hola de la segunda línea. **La razón es la forma en la que trabaja echo: pone en la salida estándar un argumento, un espacio, un argumento, un espacio, ... Entonces, cuando vuelca el argumento "mundo\n", lo siguiente que hace es insertar un espacio antes de pasar al siguiente argumento.**

9.
   - echo -e hola, mundo\\nhola, mundo
   - `hola, mundo`
     `hola, mundo`
     `user1@host:~$`
   - Es igual que el anterior, pero ahora hemos quitado el espacio entre la secuencia Escape de salto de línea y hola. En la salida vemos que el espacio al principio de la segunda línea ya no está. La razón es que ahora el segundo hola ya no es un argumento separado. En bash las líneas de comandos se procesan separando los caracteres en grupos que se llaman tokens. Los tokens están separados por los metacaracteres mencionados anteriormente, entre los que se encuentra el espacio. Así pues "mundo\nhola", constituye un único token y será un solo argumento que se lanza a la salida estándar, sin insertar espacio en medio.

10.
    - echo -e "hola, mundo\nhola, mundo"
    - `hola, mundo`
      `hola, mundo`
      `user1@host:~$`
    - Usar las comillas es otra forma de "Quoting" que afecta también al carácter Escape. Dentro de las comillas el carácter Escape "\" deja de tener la funcionalidad prevista para bash y no actúa, con lo que no se hace la sustitución de "\n" por "n". Otro efecto, no visible en la salida es que ahora el comando echo solo tiene un argumento porque todo lo que está entrecomillado es considerado como un solo token. La razón es precisamente el significado de Quoting: da a los carácteres su significado literal, lo que significa que el espacio que separa las apalabras dentro de las comillas deja de funcionar como un metacarácter para separar y todo queda en un solo token. Las dobles comillas como veremos no hacen Quoting sobre todos los carácteres, hay algunas excepciones. Las comillas simples sí hacen el Quoting sobre todos los caracteres.

11.
- echo -e 'hola, mundo\nhola, mundo'
- ```
  hola, mundo
  hola, mundo
  user1@host:~$
  ```
- Usar las comillas simples es otra forma de "Quoting" incluso más estricta que las dobles comillas por lo que la explicación del anterior caso es aplicable aquí.

12.
- echo (hola, mundo)
- ```
  -bash: syntax error near unexpected token 'hola,'
  user1@host:~$
  ```
- Aquí tenemos otro ejemplo de metacaracter. El paréntesis se utiliza en bash para abrir una subshell. Por ejemplo, si en la línea de comandos hacemos:

  ```
  ( ls )
  ```

  se ejecuta el comando ls dentro de una nueva bash, la cuál es un proceso hijo de la actual. No se puede usar directamente el mecanismo de la subshell como argumento del comando, pero se puede usar el mecanismo de expansión, por ejemplo:

  ```
  echo $( ls )
  ```

  El mecanismo de expansión es muy importante en bash y hay muchos tipos. En este caso tenemos una expansión de comandos, que se expresa con el símbolo del dolar y los paréntesis. La subshell se abre, ejecuta el comando, y la salida estándar es sustiuida en la línea de comandos de la shell llamante.
  El error de síntaxis se debe a que no se espera la ejecución directa de un comando en una subshell como argumento salvo en las expansiones.

13.
- echo \(hola, mundo\)
- ```
  (hola, mundo)
  user1@host:~$
  ```
- En este caso, el error debido al intento de abrir directamente la subshell como argumento, se subsana mediante Quoting. El mecanismo empleado para el Quoting en este caso ha sido el carácter Escape.

14.
- echo hola; mundo
- ```
  hola
  -bash: mundo: command not found
  nacho@higgs:~$
  ```
- En este caso, estamos usando la línea de comandos para ejecutar una agrupación de comandos. En realidad la línea de comandos espera en general una agrupación de comandos, en varias de sus formas. La forma más sencilla es la de varios comandos consecutivos, algo que se crea utilizando el carácter ";". El primero comando que se ejecuta es echo con el argumento hola. Este comando funciona correctamente. A continuación bash intenta ejecutar el siguiente comando de la secuencia que es mundo y no lo encuentra. Por eso muestra el error.

15.
- echo hola| mundo
- ```
  -bash: mundo: command not found
  nacho@higgs:~$
  ```
- El carácter especial "|" sirve para construir una tubería donde la salida estándar del comando precedente es utilizada como la entrada entrada estándar del comando posterior. Por eso bash está tratando de interpretar mundo como un comando, y como no lo encuentra se produce el error.

16.
- echo hola& mundo
  ```

- `[1] 46`
  `hola`
  `-bash: mundo: command not found`
  `[1]+  Done                    echo hola`
  `nacho@higgs:~$`
- El carácter especial "&" sirve para indicar que el comando que lo incluya después de todos sus argumentos, se convierta en un trabajo en segundo plano. Por eso aquí el comando echo con el argumento hola, se ejecuta como un trabajo en segundo plano. Cuando se lanza un trabajo en segundo plano, bash nos muestra el número de trabajo y el número de proceso del trabajo. Otro efecto del carácter & es que permite agrupar comandos de manera similar a ";" Por eso, bash trata de ejecutar "mundo" como un comando, y como no existe se produce el error. Además, cada vez que el intérprete de comandos vuelve al prompt, muestra un mensaje relativo a los trabajos que han cambiado de estado desde la última vez. Por eso muestra el mensaje relativo a que el trabajo 1 ha terminado.

  Por ejemplo, si ejecutamos el editor vim en bash y estando en el editor pulsamos Ctrl-Z en la terminal, el efecto es que el trabajo pasa a estado "detenido" y volvemos al prompt. Antes del prompt aparece un mensaje como este:

  `[1]+  Stopped                 vim`

17. - echo hola\& mundo
    - `hola& mundo`
      `user1@host:~$`
    - En este caso aplicamos nuevamente el mecanismo de Quoting mediante el carácter Escape \ para evitar que & sea interpretado como un carácter especial.

18. - echo hola * mundo
    - `hola  p.txt p1.cc p1.txt temp mundo`
      `user1@host:~$`
    - La salida depende de los archivos que se encuentren en el directorio de trabajo. En este caso bash utiliza el carácter especial * para hacer una expansión. El caracter especial * es sustituido por una lista de los nombres de los archivos que se encuentren en el directorio de trabajo (salvo los archivos cuyo nombre empieza por ".")

19. - echo hola * mundo
    - `hola * mundo`
      `user1@host:~$`
    - Hemos utilizado el carácter Escape para hacer Quoting al carácter especial *. Por eso, ahora no se produce la expansión.

20. - echo hola > mundo
    - `user1@host:~$`
    - Vemos que en la terminal no hay salida del comando echo. La razón es que estamos utilizando una redirección en el comando. La redirección es aplicada por bash justo antes de la ejecución del comando. En la redirección del ejemplo, la salida estándar, que normalmente es la propia terminal, pasa a ser un archivo concreto, en este caso, el archivo "mundo". Este archivo es creado y recibirá todo lo que el comando envíe a su salida estándar. Si hacemos:

      `cat mundo`

      Obtendremos por lo tanto:

      `hola`

21.
- echo < mundo
- ```
  user1@host:~$
  ```
- El comando aplica en este caso otro operador de redirección, <: este operador redirecciona la entrada estándar, que normalmente es el buffer de entrada de la terminal (el teclado casi siempre), a un archivo. La idea entonces es que lo que sea lea del archivo sea interpretado por el comando como una entrada por teclado.

  Sin embargo, vemos que a pesar de que el archivo tiene un contenido, la palabra hola, del comando anterior, echo no muestra nada más que una línea vacía. La razón es que el comando echo no utiliza la entrada estándar. Lo que estamos viendo en la salida, es lo mismo que si ejecutamos echo sin argumentos.

  En cambio, si ejecutamos:

  ```
  cat < mundo
  ```

  Obtenemos:

  ```
  hola
  user1@host:~$
  ```

  ya que cat sí utiliza la entrada estándar cuando se ejecuta sin argumentos o con el argumento "-" como veremos en el siguiente ejercicio.

□

Problema 2. En la pantalla de la terminal conectada a bash se observa lo siguiente:

```
user1@host:~$ cat -
uno dos
uno dos
tres cuatro
tres cuatro
cinco seiscinco seissiete ocho
siete ocho
nueve dieznueve diezuser1@host:~$
```

Trata de explicar la salida que observas teniendo en cuenta que:

- El usuario no ha tecleado en la línea de comandos ninguna palabra repetida.
- El usuario, además de los carácteres imprimibles que se muestran solo ha utilizado Ctrl-D y nueva línea.
- La página man de cat es:

```
NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s) to standard output.

       With no FILE, or when FILE is -, read standard input.

       -A, --show-all
```
```

```
              equivalent to -vET

       -b, --number-nonblank
              number nonempty output lines, overrides -n

       -e     equivalent to -vE

       -E, --show-ends
              display $ at end of each line

       -n, --number
              number all output lines

       -s, --squeeze-blank
              suppress repeated empty output lines

       -t     equivalent to -vT

       -T, --show-tabs
              display TAB characters as ^I

       -u     (ignored)

       -v, --show-nonprinting
              use ^ and M- notation, except for LFD and TAB

       --help display this help and exit

       --version
              output version information and exit

EXAMPLES
       cat f - g
              Output f's contents, then standard input, then g's contents.
```

Solución. Para explicar esta salida hay que tener en cuenta lo siguiente:

- Para entender que está sucediendo aquí, es necesario tener al menos una aproximación al concepto de terminal en Unix. En una primera aproximación, podemos entender una terminal como dos buffers de carácteres, uno para la entrada y otro para la salida. Además, la terminal es vista por las aplicaciones Unix como un archivo. Si estando en una terminal, tecleamos:

  ```
  tty
  ```

  podemos obtener algo como:

  ```
  /dev/pts/1
  ```

  que es el archivo en el sistema de archivos que representa a la terminal donde estamos. Las operaciones de lectura y escritura de este archivo son controladas por el driver de la terminal.

  Hay que recordar aquí también que los procesos tiene un conjunto de archivos abiertos por defecto que son:

  – Entrada estándar. Tiene asociado el descriptor de archivo 0.

- Salida estándar. Tiene asociado el descriptor de archivo 1.
- Salida error. Tiene asociado el descriptor de archivo 2

Por defecto, normalmente estos tres archivos son en realidad el archivo que representa la terminal (aunque esto se puede cambiar y de hecho se hace frecuéntemente como veremos en la asignatura)

De esta manera, cuando un proceso "lee" de la entrada estándar (hace un read para el descriptor de archivo 0), el kernel del sistema va a volcar los caracteres tecleados en el buffer de entrada de la terminal y además, por defecto, hace un efecto de eco, ya que también el driver de la terminal los muestra en pantalla. Por eso, cuando un programa nos pide que introduzcamos cosas por teclado, vemos lo que estamos tecleando. Esto está sucediendo en este comando cat que está leyendo la entrada estándar.

Por otra parte, los procesos por defecto tienen su salida estándar también en la terminal, por lo que cuando se lanza una orden de escritura sobre el descriptor de archivo 1, en realidad se escriben caracteres en el buffer de salida de la terminal, lo que supone que se visualicen en la misma.

- El comando introducido:

```
cat -
```

realiza la concatenación de archivos como ya se explicó. Normalmente puede escribirse como:

```
cat file1 file2
```

En ese caso se vuelca a la salida estándar la concatenación de los contenidos de file1 y file2.

Si no se ponen archivos en los argumentos de cat, se utiliza como archivo a la propia entrada estándar. Por tanto, por lo comentado en el punto anterior, se leerá lo que escribamos por teclado y se irá volcando a la salida estándar, del mismo modo que occure con un archivo.

Otra forma de obtener la entrada estándar es mediante el argumento "-" que en cat y en otros muchos comandos de unix significa "utiliza la entrada estándar" (ver ejemplo de la página man, donde se muestra que puede usarse en combinación con archivos).

- Ahora bien, observarán que el volcado a la salida no se produce al terminar el archivo, sino al finalizar cada línea (el carácter que vemos nada más pulsar la tecla no es impreso por cat, es el eco que el driver de la terminal produce a instancias de la interrupción detectada al pulsar la tecla, este eco incluso se puede desactivar en la configuración de la terminal). Veamos por qué esto es así.

  Los sistemas POSIX como Linux soportan dos modos de manejo de la entrada, la forma canónica o cooked y la forma no can'onica o raw. En la forma canónica, que es la habitual, la entrada no se procesa carácter a carácter, sino en un buffer de caracteres. En el caso de la terminal por ejemplo, esto significa que el buffer de entrada se empieza a llenar cuando empezamos a teclear el primer carácter y sigue llenándose sin enviar los datos al consumidor de los mismos hasta que no se produzca cierto evento, que puede ser la llegada de un carácter newline o un evento de fin de archivo (EOF).

  Así, cuando la terminal está funcionando en modo canónico, cuando ocurre un fin de línea, el buffer es enviado incluyendo el último carácter newline. Pongamos que cat es el consumidor de los datos, por lo que estará a la espera, seguramente suspendido en una llamada del sistema read() bloqueante, que se desbloqueará al recibir datos. En el caso de cat, al recibir los datos, continuará ejecutándose escribiendo la línea recibida en la salida estándar.

  Esto es lo que estamos viendo en la parte:

```
uno dos
uno dos
tres cuatro
tres cuatro
```

donde cada línea tecleada es repetida a continuación. La primera línea del par es escrita por el usuario: la terminal está leyendo lo que le proporciona el teclado mediante su driver, guardando los caracteres en un buffer y al mismo tiempo escribiendo en la salida estándar los mismos caracteres, incluyendo el salto de línea, por el eco realizado por el driver de la terminal. Al llegar al salto de línea, la terminal seguirá escribiendo en la siguiente línea.

Por su parte cat está leyendo la entrada estándar con un read bloqueado a la espera de recibir datos. En el modo canónico, la terminal vuelca el buffer de caracteres acumulados en la entrada estándar al terminar la línea. Entonces el read() se desbloquea al recibir los datos, cat continúa ejecutándose y escribe en la salida estándar (que es la propia terminal), el buffer recibido y por eso hay una repetición de la línea "uno dos" debajo de la otra. Como el carácter final del buffer es un salto de línea, la terminal continua escribiendo debajo (tres - cuatro)

- Sin embargo, vemos a continuación que el comportamiento cambia.

```
cinco seiscinco seissiete ocho
siete ocho
nueve dieznueve diezuser1@host:~$
```

Vemos que el usuario escribe "cinco seis" y que luego el texto se repite. Esto sucede porque hay otra forma de ordenar el envío del buffer de caracteres acumulados en la terminal, que es mediante el evento EOF. El evento EOF se desencadena en la terminal con la combinción de teclas Ctrl-D (normalmente es así, aunque esto es configurable). El significado de este evento en los sistemas POSIX es **terminar la transmisión**, lo que es equivalente a enviar el buffer de carácteres. Esto, al igual que el newline, provoca el desbloqueo del read en el cat que pasa a mostrarlo por la salida estándar. A diferencia de lo que sucede con el evento fin de línea, aquí no hay un salto de línea, ni tecleado por el usuario, ni en el buffer. Por eso, se repite a continuación el texto cinco seis y luego el usuario puede seguir tecleando "siete ocho" en la misma línea. En ese momento, el usuario introduce un newline, con lo que el buffer se vuelca en la siguiente línea y el usuario sigue teclando en otra línea nueve diez. El usuario vuelve a lanzar el evento EOF con Ctrl-D, con lo que el buffer se copia (segundo nueve-diez).

En ese punto, vemos que se vuelve al prompt, es decir, el comando cat ha terminado. **La razón es que la condición que cat utiliza para terminar es la recepción de un buffer vacío.** Por ejemplo, si al principio de una línea, el usuario lanza un EOF, el buffer no tendrá ningún car'acter y cat finaliza. O por ejemplo, si después de un evento EOF como el que ocurrió para que se escribiera la copia del texto "nueve-diez" se lanza otro evento EOF, el buffer se envía vacío y cat termina, que es lo que ha ocurrido aquí.

- Puedes realizar el siguiente experimento con el comando stty, que sirve para configurar la terminal:

```
stty -icanon
```

Esto establece que la terminal pasa a modo raw, ya no está en modo canónico. Ejecuta ahora el comando cat, para leer la entrada estándar. Observarás que ahora por cada carácter que escribimos, cat lo imprime a continuación, no espera al final de la línea. Es decir, el buffer se envía a cat con un solo carácter. La forma de terminar la entrada de carácteres ya nos es Ctrl-D tampoco, tendrás que terminar el proceso con Ctrl-C. Vueve a establecer el modo canónico de tu terminal con:

```
stty icanon
```

La utilidad del modo canónico o cooked en una terminal es la de acumular caracteres en una línea y tener ciertas opciones de edición para corregir errores.

Las aplicaciones actuales de edición de texto, o intérpretes de comandos como bash ya no utilizan esta forma primitiva de edición. Lo que hacen es poner la terminal en modo raw y usar alguna librería de edición de líneas más sofisticada como readline. Cuando bash lanza un comando, se preocupa de poner la terminal nuevamente en su configuración establecida, que suele ser el modo cooked, y cuando bash vuelve a tomar el control de la terminal, establece nuevamente el modo raw para manejar la línea con readline u otro sistema de edición de líneas.

□