

# EA872 Laboratório de Programação de Software Básico

## Atividade 1



**Vinícius Esperança Mantovani**

**RA 247395**

Entrega (limite): 09/08/2023, 13:00

### Exercício 1)

Para encontrarmos os shells disponíveis no sistema, devemos acessar o arquivo shells, contido em /etc, sendo etc um diretório contido diretamente no diretório-raiz do sistema. Isso porque, o arquivo shells contém arquivos que, por sua vez, são contenedores dos diversos shells existentes no sistema. Sendo assim, basta que usemos o comando “cat” para listar o conteúdo de shells, conforme: “cat /etc/shells”. Desse modo, temos acesso ao que apresenta na Figura 1.

```
% cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/bin/csh
/usr/bin/csh
/bin/tcsh
/usr/bin/tcsh
/usr/bin/screen
/usr/bin/sh
%
```

Figura 1: resposta do terminal para o comando cat /etc/shells

Desse modo, conseguimos ver todos os shells disponíveis no sistema, que são: sh, bash, rbash, dash, csh, tcsh e screen.

## Exercício 2)

Vale ressaltar, de início, que, para a execução do procedimento, foi criado o diretório bin sobre o qual se explica no roteiro da atividade (\$HOME/bin).

Para adicionar um caminho na variável PATH, devemos, inicialmente, conferir qual seu estado atual (Figura 3), para podermos conferir o funcionamento do procedimento após sua aplicação. Em seguida, usa-se o comando “setenv PATH \$HOME/bin:\$PATH” (csh ou tcsh), conforme a Figura 2 a seguir.

```
[pininsu@fedora ~]$ setenv PATH $HOME/bin:$PATH
```

Figura 2: comando setenv aplicado em PATH para adicionar caminho na variável

```
[pininsu@fedora ~]$ echo $PATH  
/home/pininsu/.local/bin:/home/pininsu/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin
```

Figura 3: Comando “echo \$PATH” e resposta do terminal (pré procedimento)

Desse modo, altera-se PATH como variável de ambiente, mantendo seu valor (todos os caminhos que ela contém) em shells-filhos, conforme se nota na Figura 3, a seguir.

```
[pininsu@fedora ~]$ echo $PATH  
/home/pininsu/bin:/home/pininsu/.local/bin:/home/pininsu/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin
```

Figura 4: Comando “echo \$PATH” e resposta do terminal (pós procedimento)

No entanto, caso encerremos o shell-pai e abramos o terminal novamente, temos o problema de não mais termos o caminho salvo em PATH. Logo, para mantermos o caminho salvo sempre, temos que escrever o comando “setenv PATH \$HOME/bin:\$PATH” no arquivo “.cshrc”. Para tanto, pode-se usar um editor de texto, de acordo com o que se mostra nas Figuras 5 e 6.

```
[pininsu@fedora ~]$ sudo nano .cshrc
```

Figura 5: chamada do editor de texto “nano” para edição do arquivo “.cshrc”

```
GNU nano 7.2  
setenv PATH $HOME/bin:$PATH
```

Figura 6: alteração do arquivo “.cshrc” por meio do editor nano

Deste modo apresentado, caso reiniciemos o shell principal (pai), continuamos com o caminho que desejamos em PATH, uma vez que o arquivo “.cshrc” contém a lista de comandos a serem executados logo ao início do processo shell.

### Exercício 3)

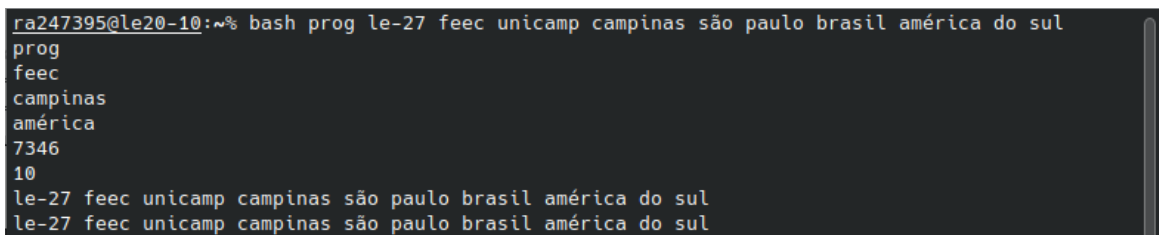
Escrevendo o programa “prog” como:

```
Unset
echo $0
echo $2
echo $4
echo $8
echo $$
echo $#
echo $*
echo $@
```

temos de resposta do shell os seguintes valores:

\$0 = prog      \$2 = feec      \$4 = campinas      \$8 = américa      \$\$ = 6268  
\$# = 10      \$\* = \$@ = le-27 feec unicamp campinas são paulo brasil américa do sul

conforme se observa na Figura 7.



```
ra247395@le20-10:~% bash prog le-27 feec unicamp campinas são paulo brasil américa do sul
prog
feec
campinas
américa
7346
10
le-27 feec unicamp campinas são paulo brasil américa do sul
le-27 feec unicamp campinas são paulo brasil américa do sul
```

Figura 7: terminal ao executar o script prog com bash

Sendo assim, a explicação para esses valores é a seguinte:

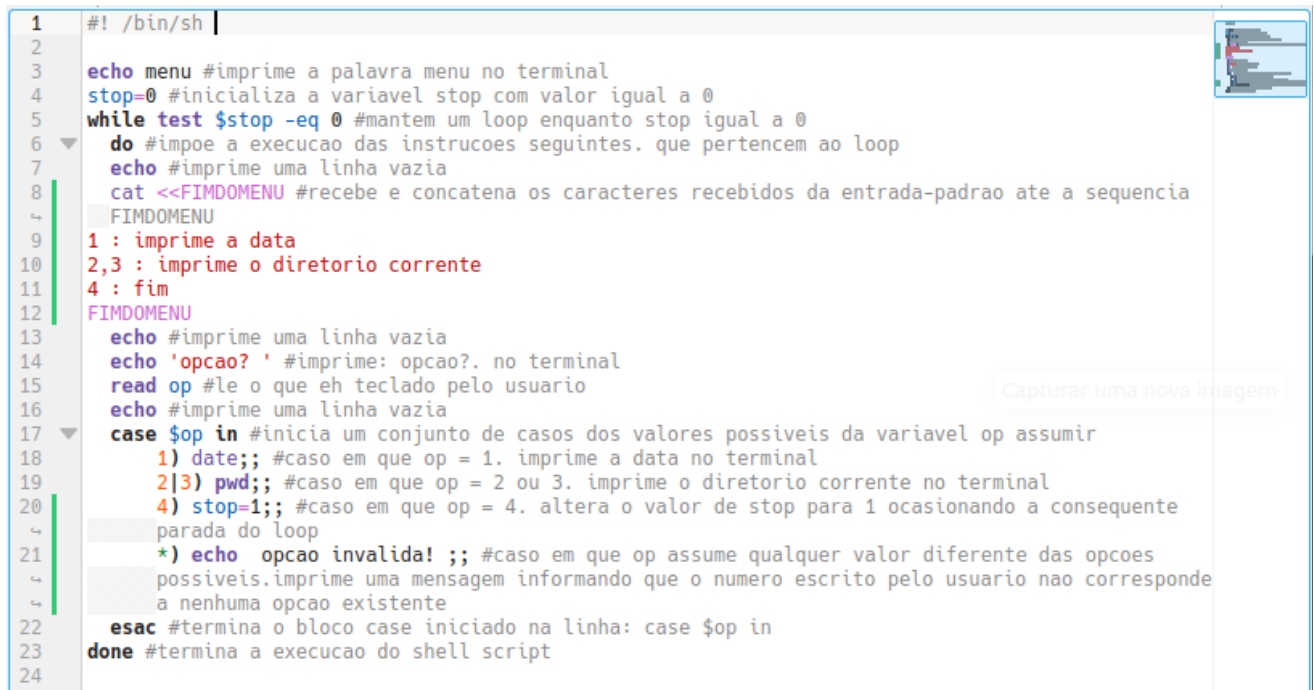
Os valores \$0, \$2, \$4 e \$8 armazenam parâmetros posicionais escritos na linha de chamada de execução do script shell. Isso, de forma que, considerando que o primeiro parâmetro posicional (“prog”) seja o parâmetro de índice 0 (nome do programa), os demais parâmetros (argumentos) são todos armazenados na sequência partindo do índice 1, ou seja: \$0 armazena parâmetro de índice 0; \$1 armazena arg de índice 1; \$2 armazena arg de índice 2; E, assim por diante.

Quanto à variável \$\$, notamos que ela assume o valor de 7346, por conta de ser esse o número de identificação do processo sh corrente. Adicionalmente, a variável \$# tem seu valor igual a 10, pois ela armazena o número de parâmetros posicionais da chamada de execução (diferentes de \$0).

Por fim, as variáveis \$\* e \$@ assumem ambas o mesmo valor (conjunto de argumentos do comando), pois as duas têm como função, armazenar os parâmetros posicionais de \$1 à \$9 do comando.

### Exercício 5)

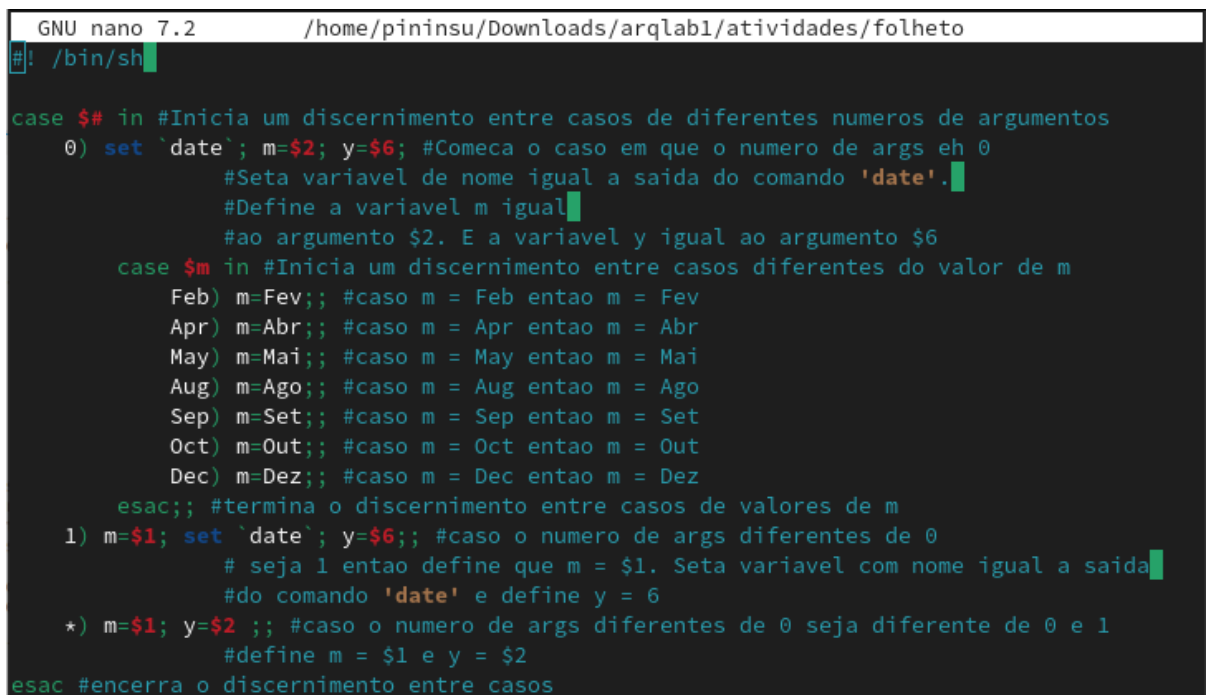
a) Os comentários feitos no script “menu” são apresentados na Figura 8 a seguir.



```
1  #!/bin/sh |
2
3  echo menu #imprime a palavra menu no terminal
4  stop=0 #inicializa a variavel stop com valor igual a 0
5  while test $stop -eq 0 #mantem um loop enquanto stop igual a 0
6  do #impoe a execucao das instrucoes seguintes. que pertencem ao loop
7      echo #imprime uma linha vazia
8      cat <<FIMDOMENU #recebe e concatena os caracteres recebidos da entrada-padrao ate a sequencia
9      FIMDOMENU
10     1 : imprime a data
11     2,3 : imprime o diretorio corrente
12     4 : fim
13     FIMDOMENU
14     echo #imprime uma linha vazia
15     echo 'opcao? ' #imprime: opcao?. no terminal
16     read op #le o que eh teclado pelo usuario
17     echo #imprime uma linha vazia
18     case $op in #inicia um conjunto de casos dos valores possiveis da variavel op assumir
19         1) date;; #caso em que op = 1. imprime a data no terminal
20         2|3) pwd;; #caso em que op = 2 ou 3. imprime o diretorio corrente no terminal
21         4) stop=1;; #caso em que op = 4. altera o valor de stop para 1 ocasionando a consequente
22         parada do loop
23         *) echo opcao invalida! ;; #caso em que op assume qualquer valor diferente das opcoes
24         possiveis.imprime uma mensagem informando que o numero escrito pelo usuario nao corresponde
25         a nenhuma opcao existente
26     esac #termina o bloco case iniciado na linha: case $op in
27 done #termina a execucao do shell script
```

Figura 8: script “menu” incrementado com comentários por linha

b) Os comentários feitos no script “folheto” são apresentados nas Figuras 9, e 10 a seguir.



```
GNU nano 7.2 /home/pininsu/Downloads/arqlab1/atividades/folheto
#!/bin/sh

case $# in #Inicia um discernimento entre casos de diferentes numeros de argumentos
0) set `date`; m=$2; y=$6; #Comeca o caso em que o numero de args eh 0
    #Seta variavel de nome igual a saida do comando 'date'.
    #Define a variavel m igual
    #ao argumento $2. E a variavel y igual ao argumento $6
    case $m in #Inicia um discernimento entre casos diferentes do valor de m
        Feb) m=Feb;; #caso m = Feb entao m = Feb
        Apr) m=Abr;; #caso m = Apr entao m = Abr
        May) m=Mai;; #caso m = May entao m = Mai
        Aug) m=Ago;; #caso m = Aug entao m = Ago
        Sep) m=Set;; #caso m = Sep entao m = Set
        Oct) m=Out;; #caso m = Oct entao m = Out
        Dec) m=Dez;; #caso m = Dec entao m = Dez
    esac;; #termina o discernimento entre casos de valores de m
1) m=$1; set `date`; y=$6;; #caso o numero de args diferentes de 0
    # seja 1 entao define que m = $1. Seta variavel com nome igual a saida
    #do comando 'date' e define y = 6
*) m=$1; y=$2 ;; #caso o numero de args diferentes de 0 seja diferente de 0 e 1
    #define m = $1 e y = $2
esac #encerra o discernimento entre casos
```

Figura 9: Script “folheto” incrementado com comentários por linha (parte 1)

```
GNU nano 7.2 /home/pininsu/Downloads/arqlab1/atividades/folheto
        #define m = $1 e y = $2
esac #encerra o discernimento entre casos
case $m in #inicia um discernimento entre casos do valor de m
jan*|Jan*) m=1;; #caso m = jan* ou Jan* entao m = 1
fev*|Fev*) m=2;; #caso m = fev* ou Fev* entao m = 2
mar*|Mar*) m=3;; #caso m = mar* ou Mar* entao m = 3
abr*|Abr*) m=4;; #caso m = abr* ou Abr* entao m = 4
mai*|Mai*) m=5;; #caso m = mai* ou Mai* entao m = 5
jun*|Jun*) m=6;; #caso m = jun* ou Jun* entao m = 6
jul*|Jul*) m=7;; #caso m = jul* ou Jul* entao m = 7
ago*|Ago*) m=8;; #caso m = ago* ou Ago* entao m = 8
set*|Set*) m=9;; #caso m = set* ou Set* entao m = 9
out*|Out*) m=10;; #caso m = out* ou Out* entao m = 10
nov*|Nov*) m=11;; #caso m = nov* ou Nov* entao m = 11
dez*|Dez*) m=12;; #caso m = dez* ou Dez* entao m = 12
[1-9]|10|11|12) ;; #caso ja seja um desses numeros entao nao faz nada

*) y=$m; m=""; #caso nao seja nenhuma dessas palavras ou desses numeros
        #entao y toma o valor de m e m passa a ser = "" (string vazia)
esac #finaliza o tratamento de casos
/usr/bin/cal $m $y #executa o comando de exibicao de calendario
```

Figura 9: Script “folheto” incrementado com comentários por linha (parte 2)

c) Os comentários feitos no script “path” são apresentados na Figura 10 a seguir.

```
GNU nano 7.2 /home/pininsu/Downloads/arqlab1/atividades/path
#!/bin/sh

for DIRPATH in `echo $PATH | sed 's:/ /g'` # Inicia um loop para cada ":" na lista
        #de caminhos em PATH
do #impoa a execucao das proximas linhas de dentro do for
    if [ ! -d $DIRPATH ] #condicional cujas expressoes sao executadas somente caso
        #$DIRPATH seja um diretorio
    then #inicia a execucao das linhas de dentro da condicional anterior
        if [ -f $DIRPATH ] #inicia uma condicional para o caso em que $DIRPATH eh um
            #arquivo simples (nao diretorio)
        then #inicia a execucao das linhas de dentro da condicional acima
            echo "$DIRPATH nao e diretorio, e um arquivo" #lanca a mensagem de aviso de
                #$DIRPATH nao eh um diretorio. Eh um arquivo simples
            else #caso contrario
                echo "$DIRPATH nao existe" #lanca a mensagem de que $DIRPATH nao existe
            fi #termina a condicional interna
        fi #termina a condicional externa
    done #termina o bloco do
```

Figura 10: Script “path” incrementado com comentários por linha

d) Os comentários feitos no script “classifica” são apresentados nas Figuras 11 e 12 a seguir.

```

GNU nano 7.2 /home/pininsu/Downloads/arqlab1/atividades/classifica Modified
#!/bin/sh

case $# in #inicia o discernimento entre casos relacionados ao numero de parametros
#posicionais diferentes do $0
0|1|[3-9]) echo Uso: classifica arquivo1 arquivo2 1>&2; exit 2 ;; #caso o numero
#de o numero de parametros posicionais (diferentes de $0) seja 0 ou 1 ou de
#3 a 9 entao imprime a mensagem "Uso: classifica arquivo1 arquivo2" no terminal
esac #encerra o bloco case
total=0; perda=0; #define as variaveis total e perda ambas com o valor 0
while read novalinha #inicia um loop que dura enquanto ha linhas a serem lidas do input
#padrao
do total=`expr $total + 1` #altera o valor da variavel total por seu proprio valor
#mais 1
case "$novalinha" in #inicia um discernimento entre casos de expressoes regulares que
#possivelmente descrevem a "novalinha" lida
*[A-Za-z]*) echo "$novalinha" >> $1 ;; #caso a "novalinha" tem em sua composicao
#um conjunto de caracteres alfabeticos. Entao escreve a
#novalinha no arquivo de caminho $1
*[0-9]*) echo "$novalinha" >> $2 ;; #caso a "novalinha" seja composta por um
#conjunto de numeros. Entao escreve a novalinha no arquivo de
#caminho $2
'<>') break;; #Caso a linha seja '<>'. Entao para (quebra) o loop

```

Figura 11: script “classifica” incrementado com comentários por linha (parte 1)

```

GNU nano 7.2 /home/pininsu/Downloads/arqlab1/atividades/classifica Modified
esac #encerra o bloco case
total=0; perda=0; #define as variaveis total e perda ambas com o valor 0
while read novalinha #inicia um loop que dura enquanto ha linhas a serem lidas do input
#padrao
do total=`expr $total + 1` #altera o valor da variavel total por seu proprio valor
#mais 1
case "$novalinha" in #inicia um discernimento entre casos de expressoes regulares que
#possivelmente descrevem a "novalinha" lida
*[A-Za-z]*) echo "$novalinha" >> $1 ;; #caso a "novalinha" tem em sua composicao
#um conjunto de caracteres alfabeticos. Entao escreve a
#novalinha no arquivo de caminho $1
*[0-9]*) echo "$novalinha" >> $2 ;; #caso a "novalinha" seja composta por um
#conjunto de numeros. Entao escreve a novalinha no arquivo de
#caminho $2
'<>') break;; #Caso a linha seja '<>'. Entao para (quebra) o loop
*) perda=`expr $perda + 1`; #caso nenhuma das possibilidades anteriores seja
#satisfeita. Entao "perda" tem seu valor incrementado em 1
esac #encerra o bloco case
done #encerra o bloco do
echo "`expr $total - 1` linha(s) lida(s), $perda linha(s) nao aproveitada(s)" #escre-
#ve o numero de linhas lidas e o numero de linhas nao aproveitadas

```

Figura 12: script “classifica” incrementado com comentários por linha (parte 2)

### Exercício 6)

Ao executar o script traps em background, recebemos como resposta no terminal o que se apresenta na Figura 13 abaixo. Essa resposta apresenta o número de vezes que traps foi executado de maneira subsequente e, em seguida, seu PID.

```
ra247395@le20-10:~% bash /home/EC21/ra247395/Documentos/atividade_1_de_872/atividades/traps &
[1] 14177
ra247395@le20-10:~% █
```

Figura 13: resposta no terminal para a execução do script traps em background

Seguindo com a análise de funcionamento, percebe-se que, ao usar ls no diretório, existe um novo arquivo criado com a execução do script (arq.14177) que apresenta o PID do processo, conforme se nota na Figura 14 a seguir.

```
ra247395@le20-10:~% ls
'Área de Trabalho'  Documentos  Imagens  Música  Público  teste2
arq.14177           Downloads  Modelos  prog    teste1  Vídeos
```

Figura 14: resposta do terminal ao comando ls após a execução do script traps

Por fim, ao usar o comando kill em seu PID (kill 14177), temos como resultado a remoção do arquivo que havia sido criado no diretório (percebe-se no uso de “ls” apresentado na Figura 16)e, no terminal, temos como resposta o que se apresenta na Figura 15 em seguida.

```
ra247395@le20-10:~$ kill 14177
ra247395@le20-10:~$ Algum processo enviou um TERM
```

Figura 15: resposta do terminal ao usar o comando kill com PID do processo gerado na execução de traps

```
ra247395@le20-10:~$ ls
'Área de Trabalho'  Downloads  Modelos  prog    teste1  Vídeos
Documentos         Imagens   Música   Público teste2
```

Figura 16: resposta no terminal ao uso de ls após a execução do comando kill 14177

Ao repetir o processo usando kill -2 <PID>, recebe-se uma resposta diferente no terminal, conforme se nota na Figura 17. Isso porque, ao invés de TERM, agora a mensagem trata do envio de um INT como se lê no terminal após a execução do comando.

```
ra247395@le20-10:~$ kill -2 15979
ra247395@le20-10:~$ Algum processo enviou um INT
█
```

Figura 17: resposta no terminal após o uso do comando kill -2 15979

Por fim, ao repetir o processo usando o comando kill -15 <PID>, recebemos uma terceira resposta do terminal, diferente da anterior, conforme a Figura 18. Neste último caso, ao contrário de INT, a mensagem trata do envio de um TERM novamente.



```
ra247395@le20-10:~% kill -15 17070
ra247395@le20-10:~% Algum processo enviou um TERM
```

Figura 18: resposta no terminal após o uso do comando kill -15 17070

Desse modo, analisando o script traps, podemos concluir que no primeiro caso, onde “kill” não foi acompanhado por argumentos, a mensagem foi igual àquela designada para o kill acompanhado de -15, pois o no script, este caso (de -15) é definido primeiro na sequência. Enquanto que a mensagem recebida ao usar o argumento -2 é diretamente definida como um caso no script traps, da mesma forma como a mensagem recebida ao usar -15, que também é diretamente definida como um caso no script.

Em conclusão, o script traps cria um arquivo arq.<PID>, define as mensagens enviadas no fechamento da execução do script e, implementa um while de espera de 5s que só para com o fim do processo (kill).

### Exercício 7)

Ao executar o comando “subspar” (executar o script subspar sem parâmetros posicionais), ocorre a saída que se vê na Figura 19 a seguir.

```
ra247395@le20-10:~$ bash /home/EC21/ra247395/Documents/atividade_1_de_872/atividades/subspar
1o resultado do teste:rs com param1 = 
2o resultado do teste:pa com param2 = pa
3o resultado do teste: com param3 = 
/home/EC21/ra247395/Documents/atividade_1_de_872/atividades/subspar: linha 11: param4: Quarto parâmetro
não iniciado
```

Figura 19: resposta no terminal para a execução do script subspar

Já ao executar o comando “subspar sp rj mg es df pr mt ms”, recebemos como resposta o que se segue na Figura 20 abaixo.

```
ra247395@le20-10:~$ bash /home/EC21/ra247395/Documents/atividade_1_de_872/atividades/subspar sp rj mg es
df pr mt ms
1o resultado do teste:sp com param1 = sp
2o resultado do teste:rj com param2 = rj
3o resultado do teste:to com param3 = mg
4o resultado do teste:es com param4 = es
```

Figura 20: resposta no terminal para a execução do comando subspar com parâmetros

Assim, para explicar as diferentes saídas, devemos entender o script. Para tanto, analisemo-no: das linhas 3 à 6 do script são estabelecidos os valores das variáveis param1, param2, param3 e param4, de forma que, elas apenas têm seus valores alterados se as variáveis, respectivamente, \$1, \$2, \$3 e \$4 obtiverem sucesso no teste para strings não nulas, ou seja, forem strings não nulas; em seguida, nas linhas de 8 à 11 são definidas as mensagens a serem apresentadas, as quais são explicadas uma a uma a seguir.

- 1) A primeira mensagem “1o resultado do teste:\${param1-rs} com param1 = \$param1” apresenta duas partes peculiares para além da string simples que a compõe. São essas: \${param1-rs} que impõe que a variável param1 será escrita nesta posição, mas caso não esteja definida, então “rs” tomará sua posição na string; e, \$param1, que impõe que o valor de \$param1 será apresentado nesta posição da string.

Seguindo a explicação 1) anterior, temos que a primeira linha de saída na Figura 19 faz sentido, porque na ausência de definição de param1, a primeira posição peculiar da string



é tomada por rs, enquanto que a segunda posição, por apresentar o conteúdo de param1, apresenta vazio na string. Já para a Figura 20, percebe-se também que a primeira linha de mensagem faz sentido, uma vez que, na primeira posição peculiar da string, quem aparece é o conteúdo de \$param1 como esperávamos e, na segunda posição, quem aparece é, de maneira igual à primeira posição, o “sp” (conteúdo de param1).

- 2) A segunda mensagem "2o resultado do teste:\${param2=pa} com param2 = \$param2", como a primeira, contém dois trechos peculiares que fogem da simplicidade da string, que são: {param2=pa} que faz com que param2 seja escrito em seu lugar, mas que caso param2 não tenha sido definido, “pa” seja escrito em seu lugar na string e seu conteúdo passe a ser “pa” (\$param2 = pa); e, \$param2 que exibe o conteúdo de \$param2 na posição em que se apresenta na string.

Seguindo a explicação 2) anterior, temos que a segunda linha de saída da Figura 19 faz sentido, porque na ausência de param2, ocorre que ambas as posições peculiares da string são preenchidas por “pa”, a primeira porque param2 não está definido e, a segunda posição porque param2 passa a ser “pa” na primeira posição. Ademais, na Figura 20, percebe-se que, como param2 é definido no início da execução do script, então ambas as posições são preenchidas com “rj” (seu conteúdo), pois na primeira posição param2 já era definido e, portanto é apresentado e não tem seu valor alterado e, por conseguinte, na segunda, mantém seu valor e tem este apresentado na string.

- 3) A terceira mensagem "3o resultado do teste:\${param3+to} com param3 = \$param3", como as duas anteriores, contém dois trechos peculiares que fogem da simplicidade da string, que são: \${param3+to} que impõe que, se param3 está definido, então “to” será exibido nessa posição da string; e, \$param3 que é substituída na impressão da string pelo valor de param3.

Seguindo a explicação 3) anterior, temos que a terceira linha de saída da Figura 19 faz sentido, pois como param3 não está definido, então ambas as partes da string que dependem de param3 são vazias, a primeira pois é vazia no caso de param3 ser indefinido e a segunda pois é substituída na string pelo valor de param3 que não é definido. Ademais, a terceira linha da saída da Figura 20 também faz sentido, uma vez que, como param3 é definido, na primeira posição peculiar, notamos que aparece “to”, conforme esperado, enquanto que na segunda, aparece, também conforme esperado, o valor de param3 (“mg”).

- 4) A quarta mensagem "4o resultado do teste:\${param4?Quarto parâmetro não iniciado} com param4 = \$param4", do mesmo modo como as anteriores, possui duas partes peculiares: \${param4?Quarto parâmetro não iniciado} que determina que se param4 não é definido, então a mensagem “Quarto parâmetro não iniciado” substitui a string que deveria ser impressa; e, \$param4 que é substituída pelo valor de param4.

Seguindo a explicação 4) anterior, temos que a quarta linha de saída da Figura 19 faz sentido, pois param4 não é definido, logo a mensagem de saída deve ser “Quarto parâmetro não iniciado” como é realmente. Além disso, notamos também a corretude da saída na Figura 20, uma vez que, como param4 é definido, na primeira posição de parte peculiar temos apresentado na string o valor de param4 e, na segunda posição também, conforme é esperado.

### Exercício 8)

Analisando de maneira sintetizada, nota-se que o script em questão implementa um diretório “\$HOME/lixo” (caso ainda não exista) e seu funcionamento. O script nos permite ver o conteúdo do diretório “\$HOME/lixo” (ls \$HOME/lixo) por meio do comando “<script> -l”, apagar o conteúdo do diretório lixo com o comando “<script> -r” e, mover itens para o diretório lixo com o comando “<script> <arquivo1> <arquivo2> ... <arquivoN>”.

Analisando mais profundamente, de maneira detalhada linha-a-linha, temos o seguinte:

- 1) test -d \$HOME/lixo || mkdir \$HOME/lixo      Testa se o diretório \$Home/lixo já existe e, caso não exista, ele é criado;
- 2) test 0 -eq "\$#" && exit 1;      Testa se o nº de parâmetros é igual a 0, caso seja, sai do shell e para a execução do script, por consequência;
- 3) case \$1 in      Passa a discernir entre casos expostos nas linhas seguintes e o que se faz em cada um deles (todos casos relacionados ao valor de \$1, ou seja, do primeiro argumento do comando, segundo parâmetro posicional);
- 4) -l) ls \$HOME/lixo;;      Primeiro caso, \$1 = -l, executa o comando ls no diretório lixo, exibindo ao usuário o conteúdo deste diretório;
- 5) -r) case \$# in      Segundo caso, \$1 = -r, inicia outra análise de casos, agora relacionada ao número de parâmetros posicionais (diferentes de \$0);
- 6) 1) aux=\$PWD; cd \$HOME/lixo; rm -rf \*; cd \$aux;;      Primeiro caso da análise do número de argumentos (diferentes de \$0) para o qual este número deve ser 1. Se for 1 de fato, apaga o conteúdo do diretório lixo usando o comando “rm -rf” (-r: remoção sequencial, -f: remoção forçada);
- 7) \*) echo pro\_lixo: Uso incorreto;;      Segundo caso da análise do número de parâmetros posicionais (diferentes de \$0), para o qual este número não é igual a 1. Neste caso, será impressa a mensagem “pro\_lixo: Uso incorreto” no terminal.
- 8) esac;;      Finaliza a análise de casos interna (relacionada ao número de parâmetros posicionais). A partir daqui continua apenas a análise de casos relacionados ao valor do primeiro argumento do comando, \$1.
- 9) \*) for i in \$\*      Inicia um loop caso o valor do primeiro argumento do comando, \$1, seja qualquer um diferente dos casos anteriores. Este loop abrange os parâmetros posicionais de \$1 à \$9 do comando;
- 10) do      inicia a execução do bloco interno do loop;
- 11) if test -f \$i      Executa os blocos seguintes caso o parâmetro \$i contenha um arquivo existente e regular;
- 12) then mv \$i \$HOME/lixo      Move o arquivo de nome contido em \$i para o diretório lixo se a condicional acima for atendida;
- 13) else echo pro\_lixo: Arquivo \$i nao encontrado.      Caso a condicional não tenha sido suprida, emite a mensagem “pro\_lixo: Arquivo \$i nao encontrado”;
- 14) fi      Termina o bloco if;
- 15) done;;      Finaliza o bloco do;
- 16) esac      Finaliza o bloco case;

Deste modo se dá o funcionamento do script em questão.

### Exercício 9)

Analisemos o script a seguir:

```
Unset
if [ $# -eq 0 ]
then
    set $PWD
fi
for ARG in $*
do
    case $ARG in
        --prof=*)
            PROFUNDIDADE=`echo $ARG | cut -f 2 -d '='`
            ;;
        *)
            if [ -d $ARG ]
            then
                CONT=${PROFUNDIDADE=0}
                while [ $CONT -gt 0 ]
                do
                    echo -n " "
                    CONT=`expr $CONT - 1`
                done
                echo "+$ARG"
                cd $ARG
                for NAME in *
                do
                    tree --prof=`expr $PROFUNDIDADE + 1` $NAME
                done
            else
                if [ -f $ARG ]
                then
                    CONT=${PROFUNDIDADE=0}
                    while [ $CONT -gt 0 ]
                    do
                        echo -n " "
                        CONT=`expr $CONT - 1`
                    done
                    echo "-$ARG"
                fi
            fi
        ;;
    esac
done
```

Vamos analisá-lo comando a comando:

if [ \$# -eq 0 ]	Caso o número de argumentos seja igual a 0
set \$PWD	Define a variável PWD
for ARG in \$*	Inicia um loop entre os argumentos passados

```

Para o caso -prof=:
    PROFUNDIDADE=`echo $ARG | cut -f 2 -d '='` Define o valor da variável
PROFUNDIDADE como igual ao valor após o "=" do argumento
Para o caso * (não -prof=*, caso contrário):
    if [ -d $ARG ]          Caso o arquivo especificado por ARG exista e seja um
diretório
        CONT=${PROFUNDIDADE=0}      Define CONT com o valor de
PROFUNDIDADE e, caso PROFUNDIDADE não esteja definida, define CONT como 0
        while [ $CONT -gt 0 ]      Loopa até o valor de CONT ser 0
            echo -n " "          Imprime um espaço em branco no terminal sem
quebra de linha
            CONT=`expr $CONT - 1`    Decrementa CONT em 1
            echo "+$ARG"          Imprime um "+" e o nome do diretório
            cd $ARG              Entra no diretório
            for NAME in *          Loopa entre os arquivos do diretório $ARG
                tree --prof=`expr $PROFUNDIDADE + 1` $NAME
            Chama o próprio script recursivamente para analisar o arquivo especificado
por NAME.
        else                    Caso não seja um diretório
            if [ -f $ARG ]        Caso exista
                CONT=${PROFUNDIDADE=0} Define CONT com o valor de
PROFUNDIDADE e, caso PROFUNDIDADE não esteja definida, define CONT como 0
                while [ $CONT -gt 0 ] Loopa até o valor de CONT ser 0
                    echo -n " "    Imprime um espaço em branco no terminal
sem quebra de linha
                    CONT=`expr $CONT - 1`    Decrementa CONT em 1
                    echo "-$ARG"          Imprime o nome do arquivo

```

Em síntese, é possível notar que este script funciona como uma espécie de explorador de diretórios, que traça, como seu nome diz, uma árvore de arquivos dentro desse diretório escolhido. Funciona com recursão, procurando todos os arquivos de dentro do diretório principal, entrando naqueles que também são diretórios para procurar por mais arquivos e mais diretórios e, passando por arquivos simples (não diretórios). Ou seja, trabalha traçando toda a hierarquia de arquivos e a estrutura de diretórios até a profundidade especificada como argumento da chamada do script.

### Exercício 10)

Tendo feito o script por meio do editor nano, sua versão final é apresentada a seguir (as explicações se encontram para baixo dele no relatório):

```
Unset
#!/bin/sh

for arg in $@
do
    caso=2 #esta variavel determina se e um atraso ou "repeticoes" a fazer
    numero_case=0 #numero de vezes que loopa sem encontrar uma opcao
    case $arg in
        --repeticoes=*)
            N=${arg#*=} #define N com o valor de repeticoes
                        #passado como argumento
            caso=0 #caso de repeticoes
            break
            ;;
        --atraso=*)
            M=${arg#*=}
            caso=1 #caso de atraso
            break
            ;;
        *)
            numero_case=`expr $numero_case + 1` #mais um
            #argumento que nao eh opcao
            ;;
    esac
done

if [ $caso -ne 2 ] #caso seja repeticao ou atraso
then
    shift $numero_case #deleta os argumentos anteriores ao relativo
                        #a opcao
    shift 1 #deleta o argumento relativo a opcao e, os proximos passam
            #a comecar do $1

    if [ $caso -eq 0 ] #caso em que eh repeticao
    then
        while [ $N -gt 0 ] #repete ate o numero de reps ser 0
        do
            eval $@ #executa o comando
            N=`expr $N - 1` #decrementa N para o funcionamento do
                            #loop
        done
    else #caso em que eh atraso
        sleep $M #espera $M segundos
        eval $@ #executa o comando
    fi
fi
```

```
fi

else #caso nao seja definida uma opcao
    eval $@ #executa o comando
fi
```

Este script tem dois blocos principais, o bloco for e o bloco if. O bloco “for”, serve para reconhecer os argumentos de entrada, ou seja, contém um bloco “case” que discerne entre as possibilidades do que são os argumentos: “-repeticoes”, “-atraso” ou algo além disso. No primeiro caso, “-repeticoes”, o valor de N (número de vezes que o comando será repetido) é setado como o valor após o “=” no argumento, o valor de “caso” (inicialmente igual a 2) passa a ser igual a 0 e, o bloco for é interrompido por um comando break. Já no caso “-atraso”, a variável M é definida com o valor igual àquilo que vem depois do “=” no argumento, a variável caso passa a valer 1 e, por meio de um break, o loop “for” é parado. Já no caso em que o argumento não é igual a nenhum dos dois anteriores, ocorre um incremento de 1 no valor da variável “numero\_case” (iniciada em 0) responsável por guardar o número de argumentos existentes antes um argumento ser igual a uma das opções. Por fim, vale destacar que, embora não seja expresso diretamente no bloco “for”, o caso em que não há opção também existe e, caso ocorra, apenas haverá a execução do comando passado como argumento, sem atraso nem repetição.

Quanto ao bloco if, cabe apontar que sua condição é que o valor da variável “caso” seja diferente de 2 para que somente seja executado o conjunto de instruções deste bloco no caso de ter sido passada uma opção como argumento. Desse modo, dentro do bloco, faz-se um shift de “numero\_case” e um de 1 para que todos os argumentos anteriores ao argumento que representa a opção, este último também, sejam deletados e o primeiro argumento passe a ser a primeira palavra que compõe o comando a ser executado. Após isso, caso satisfeita a condição de que o valor de caso seja 0, então entra-se em um bloco “if” interno ao outro, que inicia um “while” finalizado quando o valor de N chega a 0 (este valor é decrementado a cada iteração). Neste bloco “while”, o comando eval é usado com argumento \$@ para executar o comando passado pelo usuário N vezes. Após o “while”, há um “else” cujas instruções internas são sleep \$M e eval \$@, implementando, portanto, o caso de a opção selecionada ser a de atraso (sleep impões esse tempo de espera). Por fim, fechando-se o “if” maior, temos um “else” que implementa o caso em que \$caso é igual a 2, ou seja, não foi passada nenhuma opção como argumento, logo, apenas executa \$@ (eval \$@).

### Exercício 11)

Neste exercício, o script escrito é o seguinte:

```
Unset
#!/bin/sh

numero=0 #numero passado pelo terminal
aux=0 #auxilia no calculo do fatorial, assume o valor do produto
case $1 in #este bloco serve para analisar se eh a primeira chamada do
    #script ou não
    -repete) #caso usado para quando eh a segunda ou mais chamada
        numero=$2 #o valor do numero eh pego dos argumentos
        aux=$3 #o valor de aux eh pego dos argumentos
        numero=`expr $numero - 1` #o valor de numero eh decrementado
        #em um para ser passado na proxima chamada
        if [ $numero -ne 0 ] #se o numero eh diferente de 0 executa o
#bloco. Serve para caso o resultado ja tenha sido encontrado
        then
            aux=`expr $numero \* $aux` #multiplica o numero
            #atual pelo valor do produto calculado ate o momento
            bash fatorial -repete $numero $aux #chama o script
            recursivamente
        else #caso seja a chamada atual tenha sido a ultima necessaria
            echo o resultado é $3 #mensagem de resultado
        fi
        ;;
    *) #caso seja a primeira chamada (o usuario chamou)
        echo escreva o numero #mensagem para o usuario escrever
        read numero #le o numero do terminal e registra na variavel
#numero
        if [ $numero -lt 0 ] #caso seja negativo o numero passado
#pelo usuario
        then
            echo "O NUMERO NAO PODE SER NEGATIVO, EXECUTE O COMANDO>
#mensagem de erro para o caso de ser negativo
        fi
        if [ $numero -eq 0 ] #caso o valor da variavel numero seja
            #igual a 0, ou seja, o usuario escreveu 0
        then
            echo o resultado é 1 #escreve a mensagem com o resultado
        fi
        if [ $numero -gt 0 ] #caso o valor do numero seja certo (maior
            #que 0)
        then
            aux=$numero #define aux com o valor de numero
            numero=`expr $numero - 1` #decrementa numero em 1
            aux=`expr $numero \* $aux` #multiplica aux pelo numero
            #decrementado
```



```

bash fatorial -repete $numero $aux #chama o proprio
                                #script para comecar a recursao

fi
;;

esac

```

Este script funciona da seguinte maneira: Usam-se duas variáveis, “aux” e “numero”. A primeira armazena o produto do que será, na última chamada recursiva, o valor do resultado. Já a segunda variável, “numero”, armazena, inicialmente, o número que o usuário passou por meio do terminal e, durante a recursão, ele é decrementado em um a cada chamada do script para que se possa multiplicar aux por “numero” e obter o próximo produto que desencadeará, por fim, no produto final, o valor do fatorial.

Partindo dessas duas variáveis, temos um bloco “case” que discerne entre valores possíveis do primeiro argumento passado na chamada do script. Isso é feito para implementar a recursão, de forma que, para diferenciar a chamada do script pelo usuário, implementa-se a opção “-repete” que não pode ser usada pelo usuário. Sendo assim, todas as chamadas do script dentro dele próprio são acompanhadas pela opção “-repete” e, esse caso é tratado de forma que os argumentos 2 e 3 são os valores assumidos, na chamada anterior, de “numero” e “aux”. Assim, no bloco “case”, ao entrar no caso “-repete”, as variáveis “numero” e “aux” passam a assumir os valores dos argumentos respectivos a cada uma delas, em seguida, “numero” é decrementado em 1. Então, caso “numero” ainda não seja igual a 0 (não seja a última chamada do script), multiplica-se o valor de “aux” pelo valor de “numero” e, chama-se o script novamente, com os valores das variáveis passados como argumentos após “-repete”. Caso a chamada atual tenha sido a última necessária e “numero” já valha 0, então é executado um “echo” com uma mensagem contenedora do resultado final.

Há também, o caso no bloco case que trata da primeira chamada, aquela feita pelo usuário. Neste caso, é impressa uma mensagem no terminal pedindo para o usuário escrever o número que deseja inserir e, já em seguida, existe uma condicional tratando do caso de ele inserir um número negativo, para o qual é impressa uma mensagem de erro. Após isso, existe outra condicional tratando do caso em que o número passado seja o 0, para o qual o valor do fatorial deve ser, de prontidão, igual a 1. Por fim, caso o valor do número seja maior que 0, entramos em uma condicional cujo primeiro comando define o “aux” com o valor de “numero”. Em seguida, ainda na condicional, o valor de “numero” é decrementado em 1 e “aux” passa a assumir o valor do produto de “aux” com “numero”. Finalmente, inicia-se a recursão, com a primeira chamada do script pelo próprio script.

O algoritmo, em síntese, funciona assim:

```

numero = numero passado no terminal
aux = numero;
numero = numero - 1
aux = aux*numero
numero = numero - 1
aux = aux*numero
.
.

```

```
aux = aux*1
resultado = aux
FIM
```

A seguir estão exemplos da execução:

Número bom:

```
[pininsu@MiWiFi-RA72-srv Documents]$ bash fatorial
escreva o numero
5
o resultado é 120
```

Figura 21: teste do script fatorial para um número determinável

Número 0:

```
[pininsu@MiWiFi-RA72-srv Documents]$ bash fatorial
escreva o numero
0
o resultado é 1
```

Figura 22: teste do script fatorial para o número 0

Número negativo:

```
[pininsu@MiWiFi-RA72-srv Documents]$ bash fatorial
escreva o numero
-10
O NUMERO NAO PODE SER NEGATIVO, EXECUTE O COMANDO NOVAMENTE
```

Figura 23: teste do script fatorial para um número negativo

Número muito grande:

```
[pininsu@MiWiFi-RA72-srv Documents]$ bash fatorial
escreva o numero
1000000000000
bash: warning: shell level (1000) too high, resetting to 1
```

Figura 24: teste do script fatorial para um número muito maior que o limite computacional