**Canivete Suíço do Shell (Bash)**

[**Aurelio Marinho Jargas**](http://aurelio.net)

**Desde 2003**

|  |  |
| --- | --- |
| [http://aurelio.net/img/icon/128/livroshell.png](http://www.shellscript.com.br/) | Este Canivete Suíço do Shell é um apêndice do meu livro [Shell Script Profissional](http://www.shellscript.com.br). O livro é enorme, são quase 500 páginas de puro shell, explicado de maneira clara e didática. Você vai dominar o assunto, além de aprender a escrever programas de verdade em vez de meros scripts toscos. Sou escritor e ganho a vida com meus livros, então comprando o livro você também me ajuda a manter esse site no ar e assim eu continuo escrevendo para você ler. Combinado? :) |

1. [Operadores](http://aurelio.net/shell/canivete/#operadores)
2. [Redirecionamento](http://aurelio.net/shell/canivete/#redir)
3. [Variáveis especiais](http://aurelio.net/shell/canivete/#variaveis)
4. [Expansão de variáveis](http://aurelio.net/shell/canivete/#expansao)
5. [Blocos e agrupamentos](http://aurelio.net/shell/canivete/#blocos)
6. [if, for, select, while, until, case](http://aurelio.net/shell/canivete/#ifwhilefor)
7. [Opções do comando test ou [](http://aurelio.net/shell/canivete/#test)
8. [Escapes especiais para usar no prompt (PS1)](http://aurelio.net/shell/canivete/" \l "prompt)
9. [Escapes reconhecidos pelo comando echo](http://aurelio.net/shell/canivete/#echo)
10. [Formatadores do comando date](http://aurelio.net/shell/canivete/#date)
11. [Formatadores do comando printf](http://aurelio.net/shell/canivete/#printf)
12. [Letras identificadoras de arquivos no comando ls -l](http://aurelio.net/shell/canivete/#ls)
13. [Curingas para nomes de arquivo (glob)](http://aurelio.net/shell/canivete/#glob)
14. [Curingas para os itens do comando case](http://aurelio.net/shell/canivete/#case)
15. [Sinais para usar com trap/kill/killall](http://aurelio.net/shell/canivete/#sinais)
16. [Códigos de retorno de comandos](http://aurelio.net/shell/canivete/#exitcode)
17. [Códigos de cores (ANSI)](http://aurelio.net/shell/canivete/#cores)
18. [Os metacaracteres das expressões regulares](http://aurelio.net/shell/canivete/#metacaracteres)
19. [Metacaracteres que são diferentes nos aplicativos](http://aurelio.net/shell/canivete/#metadif)
20. [Caracteres ASCII imprimíveis (ISO-8859-1) - texto](http://aurelio.net/shell/canivete/#ascii)
21. [Caracteres ASCII imprimíveis (ISO-8859-1) - imagem](http://aurelio.net/shell/canivete/#asciiimg)
22. [Códigos prontos para copiar e colar](http://aurelio.net/shell/canivete/#codigos)
23. [Atalhos da linha de comando (set -o emacs)](http://aurelio.net/shell/canivete/#emacs)
24. [A caixa de ferramentas do shelleiro](http://aurelio.net/shell/canivete/#ferramentas)

* Quer imprimir o canivete? [Baixe o PDF](http://aurelio.net/shell/canivete/pdf/).
* Veja também: [miniman](http://aurelio.net/shell/miniman/), o irmãozinho do canivete suíço.
* Veja o [código-fonte do canivete](http://aurelio.net/shell/canivete/index.t2t), no formato [txt2tags](http://txt2tags.org/pt/).
* Traduções do canivete: [Español](http://aurelio.net/shell/canivete/es/), [English](http://aurelio.net/shell/canivete/en/)

**1. Operadores**

|  |  |
| --- | --- |
| **Operadores Aritméticos** | |
| + | Adição |
| - | Subtração |
| \* | Multiplicação |
| / | Divisão |
| % | Módulo |
| \*\* | Exponenciação |
| **Operadores de Atribuição** | |
| = | Atribui valor a uma variável |
| += | Incrementa a variável por uma constante |
| -= | Decrementa a variável por uma constante |
| \*= | Multiplica a variável por uma constante |
| /= | Divide a variável por uma constante |
| %= | Resto da divisão por uma constante |
| ++ | Incrementa em 1 o valor da variável |
| -- | Decrementa em 1 o valor da variável |
| **Operadores Relacionais** | |
| == | Igual |
| != | Diferente |
| > | Maior |
| >= | Maior ou Igual |
| < | Menor |
| <= | Menor ou Igual |
| **Operadores Lógicos** | |
| && | E lógico (AND) |
| || | OU lógico (OR) |
| **Operadores de BIT** | |
| << | Deslocamento à esquerda |
| >> | Deslocamento à direita |
| & | E de bit (AND) |
| | | OU de bit (OR) |
| ^ | OU exclusivo de bit (XOR) |
| ~ | Negação de bit |
| ! | NÃO de bit (NOT) |
| **Operadores de BIT (atribuição)** | |
| <<= | Deslocamento à esquerda |
| >>= | Deslocamento à direita |
| &= | E de bit |
| |= | OU de bit |
| ^= | OU exclusivo de bit |

**2. Redirecionamento**

|  |  |
| --- | --- |
| **Operador** | **Ação** |
| < | Redireciona a entrada padrão (STDIN) |
| > | Redireciona a saída padrão (STDOUT) |
| 2> | Redireciona a saída de erro (STDERR) |
| >> | Redireciona a saída padrão, anexando |
| 2>> | Redireciona a saída de erro, anexando |
| | | Conecta a saída padrão com a entrada padrão de outro comando |
| 2>&1 | Conecta a saída de erro na saída padrão |
| >&2 | Conecta a saída padrão na saída de erro |
| >&- | Fecha a saída padrão |
| 2>&- | Fecha a saída de erro |
| 3<>arq | Conecta o descritor de arquivos 3 ao arquivo 'arq' |
| <<FIM | Alimenta a entrada padrão (Here Document) |
| <<-FIM | Alimenta a entrada padrão, cortando TABs |
| <(cmd) | A saída do comando 'cmd' é um arquivo: diff <(cmd1) <(cmd2) |
| >(cmd) | A entrada do comando 'cmd' é um arquivo: tar cf >(bzip2 -c >file.tbz) $dir |

**3. Variáveis especiais**

|  |  |
| --- | --- |
| **Variável** | **Parâmetros Posicionais** |
| $0 | Parâmetro número 0 (nome do comando ou função) |
| $1 | Parâmetro número 1 (da linha de comando ou função) |
| ... | Parâmetro número N ... |
| $9 | Parâmetro número 9 (da linha de comando ou função) |
| ${10} | Parâmetro número 10 (da linha de comando ou função) |
| ... | Parâmetro número NN ... |
| $# | Número total de parâmetros da linha de comando ou função |
| $\* | Todos os parâmetros, como uma string única |
| $@ | Todos os parâmetros, como várias strings protegidas |
| **Variável** | **Miscelânia** |
| $$ | Número PID do processo atual (do próprio script) |
| $! | Número PID do último job em segundo plano |
| $\_ | Último argumento do último comando executado |
| $? | Código de retorno do último comando executado |

**4. Expansão de variáveis**

|  |  |
| --- | --- |
| **Sintaxe** | **Expansão Condicional** |
| ${var:-texto} | Se var não está definida, retorna 'texto' |
| ${var:=texto} | Se var não está definida, defina-a com 'texto' |
| ${var:?texto} | Se var não está definida, retorna o erro 'texto' |
| ${var:+texto} | Se var está definida, retorna 'texto', senão retorna o vazio |
| **Sintaxe** | **Expansão de Strings** |
| ${var} | É o mesmo que $var, porém não ambíguo |
| ${#var} | Retorna o tamanho da string |
| ${!var} | Executa o conteúdo de $var (igual 'eval \$$var') |
| ${!texto\*} | Retorna os nomes de variáveis começadas por 'texto' |
| ${var:N} | Retorna o texto a partir da posição 'N' |
| ${var:N:tam} | Retorna 'tam' caracteres a partir da posição 'N' |
| ${var#texto} | Corta 'texto' do início da string |
| ${var##texto} | Corta 'texto' do início da string (\* guloso) |
| ${var%texto} | Corta 'texto' do final da string |
| ${var%%texto} | Corta 'texto' do final da string (\* guloso) |
| ${var/texto/novo} | Substitui 'texto' por 'novo', uma vez |
| ${var//texto/novo} | Substitui 'texto' por 'novo', sempre |
| ${var/#texto/novo} | Se a string começar com 'texto', substitui 'texto' por 'novo' |
| ${var/%texto/novo} | Se a string terminar com 'texto', substitui 'texto' por 'novo' |

**5. Blocos e agrupamentos**

|  |  |  |
| --- | --- | --- |
| **Sintaxe** | **Descrição** | **Exemplo** |
| "..." | Protege uma string, mas reconhece $, \ e ` como especiais | "abc" |
| '...' | Protege uma string completamente (nenhum caractere é especial) | 'abc' |
| $'...' | Protege uma string completamente, mas interpreta \n, \t, \a, etc | $'abc\n' |
| `...` | Executa comandos numa subshell, retornando o resultado | `ls` |
| {...} | Agrupa comandos em um bloco | { ls ; } |
| (...) | Executa comandos numa subshell | ( ls ) |
| $(...) | Executa comandos numa subshell, retornando o resultado | $( ls ) |
| ((...)) | Testa uma operação aritmética, retornando 0 ou 1 | ((5 > 3)) |
| $((...)) | Retorna o resultado de uma operação aritmética | $((5+3)) |
| [...] | Testa uma expressão, retornando 0 ou 1 (alias do comando 'test') | [ 5 -gt 3 ] |
| [[...]] | Testa uma expressão, retornando 0 ou 1 (podendo usar && e ||) | [[ 5 > 3 ]] |

**6. if, for, select, while, until, case**

|  |  |  |  |
| --- | --- | --- | --- |
| **if** | **for / select** | **while / until** | **case** |
| if COMANDO  then  ...  elif COMANDO  then  ...  else  ...  fi | for VAR in LISTA  do  ...  done  ou:  for ((exp1;exp2;exp3)) | while COMANDO  do  ...  done | case $VAR in  txt1) ... ;;  txt2) ... ;;  txtN) ... ;;  \*) ... ;;  esac |

**7. Opções do comando test ou [**

|  |  |
| --- | --- |
| **Comparação Numérica** | |
| -lt | É menor que (LessThan) |
| -gt | É maior que (GreaterThan) |
| -le | É menor igual (LessEqual) |
| -ge | É maior igual (GreaterEqual) |
| -eq | É igual (EQual) |
| -ne | É diferente (NotEqual) |
| **Comparação de Strings** | |
| = | É igual |
| != | É diferente |
| -n | É não nula |
| -z | É nula |
| **Operadores Lógicos** | |
| ! | NÃO lógico (NOT) |
| -a | E lógico (AND) |
| -o | OU lógico (OR) |
| **Testes em arquivos** | |
| -b | É um dispositivo de bloco |
| -c | É um dispositivo de caractere |
| -d | É um diretório |
| -e | O arquivo existe |
| -f | É um arquivo normal |
| -g | O bit SGID está ativado |
| -G | O grupo do arquivo é o do usuário atual |
| -k | O sticky-bit está ativado |
| -L | O arquivo é um link simbólico |
| -O | O dono do arquivo é o usuário atual |
| -p | O arquivo é um named pipe |
| -r | O arquivo tem permissão de leitura |
| -s | O tamanho do arquivo é maior que zero |
| -S | O arquivo é um socket |
| -t | O descritor de arquivos N é um terminal |
| -u | O bit SUID está ativado |
| -w | O arquivo tem permissão de escrita |
| -x | O arquivo tem permissão de execução |
| -nt | O arquivo é mais recente (NewerThan) |
| -ot | O arquivo é mais antigo (OlderThan) |
| -ef | O arquivo é o mesmo (EqualFile) |

**8. Escapes especiais para usar no prompt (PS1)**

|  |  |  |
| --- | --- | --- |
| **Escape** | **Lembrete** | **Expande para...** |
| \a | *Alerta* | Alerta (bipe) |
| \d | *Data* | Data no formato "Dia-da-semana Mês Dia" (Sat Jan 15) |
| \e | *Escape* | Caractere Esc |
| \h | *Hostname* | Nome da máquina sem o domínio (dhcp11) |
| \H | *Hostname* | Nome completo da máquina (dhcp11.empresa) |
| \j | *Jobs* | Número de jobs ativos |
| \l | *Tty* | Nome do terminal corrente (ttyp1) |
| \n | *Newline* | Linha nova |
| \r | *Return* | Retorno de carro |
| \s | *Shell* | Nome do shell (basename $0) |
| \t | *Time* | Horário no formato 24 horas HH:MM:SS |
| \T | *Time* | Horário no formato 12 horas HH:MM:SS |
| \@ | *At* | Horário no formato 12 horas HH:MM am/pm |
| \A | *At* | Horário no formato 24 horas HH:MM |
| \u | *Usuário* | Login do usuário corrente |
| \v | *Versão* | Versão do Bash (2.00) |
| \V | *Versão* | Versão+subversão do Bash (2.00.0) |
| \w | *Working Dir* | Diretório corrente, caminho completo ($PWD) |
| \W | *Working Dir* | Diretório corrente, somente o último (basename $PWD) |
| \! | *Histórico* | Número do comando corrente no histórico |
| \# | *Número* | Número do comando corrente |
| \$ | *ID* | Mostra "#" se for root, "$" se for usuário normal |
| \nnn | *Octal* | Caractere cujo octal é nnn |
| \\ | *Backslash* | Barra invertida \ literal |
| \[ | *Escapes* | Inicia uma seqüência de escapes (tipo códigos de cores) |
| \] | *Escapes* | Termina uma seqüência de escapes |

**9. Escapes reconhecidos pelo comando echo**

|  |  |  |
| --- | --- | --- |
| **Escape** | **Lembrete** | **Descrição** |
| \a | *Alerta* | Alerta (bipe) |
| \b | *Backspace* | Caractere Backspace |
| \c | *EOS* | Termina a string |
| \e | *Escape* | Caractere Esc |
| \f | *Form feed* | Alimentação |
| \n | *Newline* | Linha nova |
| \r | *Return* | Retorno de carro |
| \t | *Tab* | Tabulação horizontal |
| \v | *Vtab* | Tabulação vertical |
| \\ | *Backslash* | Barra invertida \ literal |
| \nnn | *Octal* | Caractere cujo octal é nnn |
| \xnn | *Hexa* | Caractere cujo hexadecimal é nn |

**10. Formatadores do comando date**

|  |  |
| --- | --- |
| **Formato** | **Descrição** |
| %a | Nome do dia da semana abreviado (Dom..Sáb) |
| %A | Nome do dia da semana (Domingo..Sábado) |
| %b | Nome do mês abreviado (Jan..Dez) |
| %B | Nome do mês (Janeiro..Dezembro) |
| %c | Data completa (Sat Nov 04 12:02:33 EST 1989) |
| %y | Ano (dois dígitos) |
| %Y | Ano (quatro dígitos) |
| %m | Mês (01..12) |
| %d | Dia (01..31) |
| %j | Dia do ano (001..366) |
| %H | Horas (00..23) |
| %M | Minutos (00..59) |
| %S | Segundos (00..60) |
| %s | Segundos desde 1º de Janeiro de 1970 |
| %% | Um % literal |
| %t | Um TAB |
| %n | Uma quebra de linha |

**11. Formatadores do comando printf**

|  |  |
| --- | --- |
| **Formato** | **Descrição** |
| %d | Número decimal |
| %o | Número octal |
| %x | Número hexadecimal (a-f) |
| %X | Número hexadecimal (A-F) |
| %f | Número com ponto flutuante |
| %e | Número em notação científica (e+1) |
| %E | Número em notação científica (E+1) |
| %s | String |

**12. Letras identificadoras de arquivos no comando ls -l**

|  |  |  |
| --- | --- | --- |
| **Letra** | **Lembrete** | **Tipos de Arquivo (primeiro caractere)** |
| - | - | Arquivo normal |
| d | Directory | Diretório |
| l | Link | Link simbólico |
| b | Block | Dispositivo de blocos (HD) |
| c | Char | Dispositivo de caracteres (modem serial) |
| s | Socket | Socket mapeado em arquivo (comunicação de processos) |
| p | Pipe | FIFO ou Named Pipe (comunicação de processos) |
| **Letra** | **Lembrete** | **Permissões do Arquivo (próximos nove caracteres)** |
| - | - | Permissão desativada |
| r | Read | Acesso de leitura |
| w | Write | Acesso de escrita |
| x | eXecute | Acesso de execução (ou acesso ao diretório) |
| X | eXecute | Acesso ao diretório somente |
| s | Set ID | Usuário/grupo para execução (SUID, SGID) - permissão 'x' ativada |
| S | Set ID | Usuário/grupo para execução (SUID, SGID) - permissão 'x' desativada |
| t | sTicky | Usuários só apagam seus próprios arquivos - permissão 'x' ativada |
| T | sTicky | Usuários só apagam seus próprios arquivos - permissão 'x' desativada |

**13. Curingas para nomes de arquivo (glob)**

|  |  |  |
| --- | --- | --- |
| **Curinga** | **Casa com...** | **Exemplo** |
| \* | Qualquer coisa | \*.txt |
| ? | Um caractere qualquer | arquivo-??.zip |
| [...] | Qualquer um dos caracteres listados | [Aa]rquivo.txt |
| [^...] | Qualquer um caractere, exceto os listados | [^A-Z]\*.txt |
| {...} | Qualquer um dos textos separados por vírgula | arquivo.{txt,html} |

**14. Curingas para os itens do comando case**

|  |  |  |
| --- | --- | --- |
| **Curinga** | **Casa com...** | **Exemplo** |
| \* | Qualquer coisa | \*.txt) echo ;; |
| ? | Um caractere qualquer | arquivo-??.zip) echo ;; |
| [...] | Qualquer um dos caracteres listados | [0-9]) echo ;; |
| [^...] | Qualquer um caractere, exceto os listados | [^0-9]) echo ;; |
| ...|... | Qualquer um dos textos separados por | | txt|html) echo ;; |

**15. Sinais para usar com trap/kill/killall**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **#** | **Linux** | **Cygwin** | **SystemV** | **AIX** | **HP-UX** | **Solaris** | **BSD/Mac** |
| 1 | HUP | HUP | HUP | HUP | HUP | HUP | HUP |
| 2 | INT | INT | INT | INT | INT | INT | INT |
| 3 | QUIT | QUIT | QUIT | QUIT | QUIT | QUIT | QUIT |
| 4 | ILL | ILL | ILL | ILL | ILL | ILL | ILL |
| 5 | TRAP | TRAP | TRAP | TRAP | TRAP | TRAP | TRAP |
| 6 | ABRT | ABRT | IOT | LOST | ABRT | ABRT | ABRT |
| 7 | BUS | EMT | EMT | EMT | EMT | EMT | EMT |
| 8 | FPE | FPE | FPE | FPE | FPE | FPE | FPE |
| 9 | KILL | KILL | KILL | KILL | KILL | KILL | KILL |
| 10 | USR1 | BUS | BUS | BUS | BUS | BUS | BUS |
| 11 | SEGV | SEGV | SEGV | SEGV | SEGV | SEGV | SEGV |
| 12 | USR2 | SYS | SYS | SYS | SYS | SYS | SYS |
| **#** | **Linux** | **Cygwin** | **SystemV** | **AIX** | **HP-UX** | **Solaris** | **BSD/Mac** |
| 13 | PIPE | PIPE | PIPE | PIPE | PIPE | PIPE | PIPE |
| 14 | ALRM | ALRM | ALRM | ALRM | ALRM | ALRM | ALRM |
| 15 | TERM | TERM | TERM | TERM | TERM | TERM | TERM |
| 16 | - | URG | USR1 | URG | USR1 | USR1 | URG |
| 17 | CHLD | STOP | USR2 | STOP | USR2 | USR2 | STOP |
| 18 | CONT | TSTP | CHLD | TSTP | CHLD | CHLD | TSTP |
| 19 | STOP | CONT | PWR | CONT | PWR | PWR | CONT |
| 20 | TSTP | CHLD | WINCH | CHLD | VTALRM | WINCH | CHLD |
| 21 | TTIN | TTIN | URG | TTIN | PROF | URG | TTIN |
| 22 | TTOU | TTOU | IO | TTOU | IO | IO | TTOU |
| 23 | URG | IO | STOP | IO | WINCH | STOP | IO |
| 24 | XCPU | XCPU | TSTP | XCPU | STOP | TSTP | XCPU |
| **#** | **Linux** | **Cygwin** | **SystemV** | **AIX** | **HP-UX** | **Solaris** | **BSD/Mac** |
| 25 | XFSZ | XFSZ | CONT | XFSZ | TSTP | CONT | XFSZ |
| 26 | VTALRM | VTALRM | TTIN | - | CONT | TTIN | VTALRM |
| 27 | PROF | PROF | TTOU | MSG | TTIN | TTOU | PROF |
| 28 | WINCH | WINCH | VTALRM | WINCH | TTOU | VTALRM | WINCH |
| 29 | IO | LOST | PROF | PWR | URG | PROF | INFO |
| 30 | PWR | USR1 | XCPU | USR1 | LOST | XCPU | USR1 |
| 31 | SYS | USR2 | XFSZ | USR2 | - | XFSZ | USR2 |
| 32 | - | - | - | PROF | - | WAITING | - |
| 33 | - | - | - | DANGER | - | LWP | - |
| 34 | - | - | - | VTALRM | - | FREEZE | - |
| 35 | - | - | - | MIGRATE | - | THAW | - |
| 36 | - | - | - | PRE | - | CANCEL | - |
| 37 | - | - | - | - | - | LOST | - |

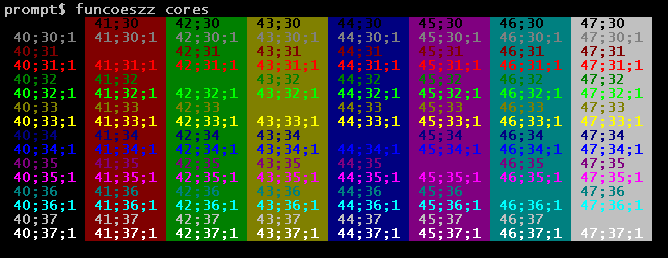
Como obter a listagem: trap -l, kill -l ou killall -l  
Veja também: [man 7 signal](http://unixhelp.ed.ac.uk/CGI/man-cgi?signal+7)

**16. Códigos de retorno de comandos**

|  |  |  |
| --- | --- | --- |
| **Código** | **Significado** | **Exemplo** |
| 0 | Nenhum erro, execução terminou OK | echo |
| 1 | A maioria dos erros comuns na execução | echo $((1/0)) |
| 2 | Erro de uso em algum 'builtin' do Shell | - |
| 126 | Comando não executável (sem permissão) | touch a ; ./a |
| 127 | Comando não encontrado ("command not found") | echooo |
| 128 | O parâmetro para o 'exit' não é um decimal | exit 1.0 |
| 128+n | 128 + código do sinal que o matou | kill -9 $PPID #exit 137 |
| 130 | O programa interrompido com o Ctrl+C (128 + 2) | - |
| 255 | Parâmetro para o 'exit' não está entre 0 e 255 | exit -1 |

**17. Códigos de cores (ANSI)**

|  |  |  |
| --- | --- | --- |
| **Cor** | **Letra** | **Fundo** |
| Preto | 30 | 40 |
| Vermelho | 31 | 41 |
| Verde | 32 | 42 |
| Amarelo | 33 | 43 |
| Azul | 34 | 44 |
| Rosa | 35 | 45 |
| Ciano | 36 | 46 |
| Branco | 37 | 47 |
| **Atributo** | | **Valor** |
| Reset | | 0 |
| Negrito | | 1 |
| Sublinhado | | 4 |
| Piscando | | 5 |
| Reverso | | 7 |
| **Exemplos: ESC [ <N>;<N> m** | | |
| Texto normal (desliga cores) | | ESC[m |
| Negrito | | ESC[1m |
| Amarelo | | ESC[33;1m |
| Fundo azul, letra cinza | | ESC[44;37m |
| Vermelho piscando | | ESC[31;5m |
| **Na linha de comando** | | |
| echo -e '\e[33;1m amarelo \e[m' | | |
| echo -e '\033[33;1m amarelo \033[m' | | |



**18. Os metacaracteres das expressões regulares**

|  |  |  |
| --- | --- | --- |
| **Meta** | **Nome** | **Descrição** |
| . | Ponto | Curinga de um caractere |
| [] | Lista | Casa qualquer um dos caracteres listados |
| [^] | Lista negada | Casa qualquer caractere, exceto os listados |
| ? | Opcional | O item anterior pode aparecer ou não (opcional) |
| \* | Asterisco | O item anterior pode aparecer em qualquer quantidade |
| + | Mais | O item anterior deve aparecer no mínimo uma vez |
| {,} | Chaves | O item anterior deve aparecer na quantidade indicada {mín,máx} |
| ^ | Circunflexo | Casa o começo da linha |
| $ | Cifrão | Casa o fim da linha |
| \b | Borda | Limita uma palavra (letras, números e sublinhado) |
| \ | Escape | Escapa um meta, tirando seu poder |
| | | Ou | Indica alternativas (usar com o grupo) |
| () | Grupo | Agrupa partes da expressão, é quantificável e multinível |
| \1 | Retrovisor | Recupera o conteúdo do grupo 1 |
| \2 | Retrovisor | Recupera o conteúdo do grupo 2 (segue até o \9) |
| .\* | Curinga | Casa qualquer coisa, é o tudo e o nada |
| ?? | Opcional NG | Idem ao opcional comum, mas casa o mínimo possível |
| \*? | Asterisco NG | Idem ao asterisco comum, mas casa o mínimo possível |
| +? | Mais NG | Idem ao mais comum, mas casa o mínimo possível |
| {}? | Chaves NG | Idem às chaves comuns, mas casa o mínimo possível |

**19. Metacaracteres que são diferentes nos aplicativos**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Programa** | **Opc** | **Mais** | **Chaves** | **Borda** | **Ou** | **Grupo** |
| awk | ? | + | - | - | | | () |
| ed | \? | \+ | \{,\} | \b | \| | \(\) |
| egrep | ? | + | {,} | \b | | | () |
| emacs | ? | + | - | \b | \| | \(\) |
| expect | ? | + | - | - | | | () |
| find | ? | + | - | \b | \| | \(\) |
| gawk | ? | + | {,} | \<\> | | | () |
| grep | \? | \+ | \{,\} | \b | \| | \(\) |
| mawk | ? | + | - | - | | | () |
| perl | ? | + | {,} | \b | | | () |
| php | ? | + | {,} | \b | | | () |
| python | ? | + | {,} | \b | | | () |
| sed | \? | \+ | \{,\} | \<\> | \| | \(\) |
| vim | \= | \+ | \{,} | \<\> | \| | \(\) |

**20. Caracteres ASCII imprimíveis (ISO-8859-1) - texto**

$ [zzascii](http://funcoeszz.net/todas.html)

32 64 @ 96 ` 162 ¢ 194 Â 226 â

33 ! 65 A 97 a 163 £ 195 Ã 227 ã

34 " 66 B 98 b 164 ¤ 196 Ä 228 ä

35 # 67 C 99 c 165 ¥ 197 Å 229 å

36 $ 68 D 100 d 166 ¦ 198 Æ 230 æ

37 % 69 E 101 e 167 § 199 Ç 231 ç

38 & 70 F 102 f 168 ¨ 200 È 232 è

39 ' 71 G 103 g 169 © 201 É 233 é

40 ( 72 H 104 h 170 ª 202 Ê 234 ê

41 ) 73 I 105 i 171 « 203 Ë 235 ë

42 \* 74 J 106 j 172 ¬ 204 Ì 236 ì

43 + 75 K 107 k 173 205 Í 237 í

44 , 76 L 108 l 174 ® 206 Î 238 î

45 - 77 M 109 m 175 ¯ 207 Ï 239 ï

46 . 78 N 110 n 176 ° 208 Ð 240 ð

47 / 79 O 111 o 177 ± 209 Ñ 241 ñ

48 0 80 P 112 p 178 ² 210 Ò 242 ò

49 1 81 Q 113 q 179 ³ 211 Ó 243 ó

50 2 82 R 114 r 180 ´ 212 Ô 244 ô

51 3 83 S 115 s 181 µ 213 Õ 245 õ

52 4 84 T 116 t 182 ¶ 214 Ö 246 ö

53 5 85 U 117 u 183 · 215 × 247 ÷

54 6 86 V 118 v 184 ¸ 216 Ø 248 ø

55 7 87 W 119 w 185 ¹ 217 Ù 249 ù

56 8 88 X 120 x 186 º 218 Ú 250 ú

57 9 89 Y 121 y 187 » 219 Û 251 û

58 : 90 Z 122 z 188 ¼ 220 Ü 252 ü

59 ; 91 [ 123 { 189 ½ 221 Ý 253 ý

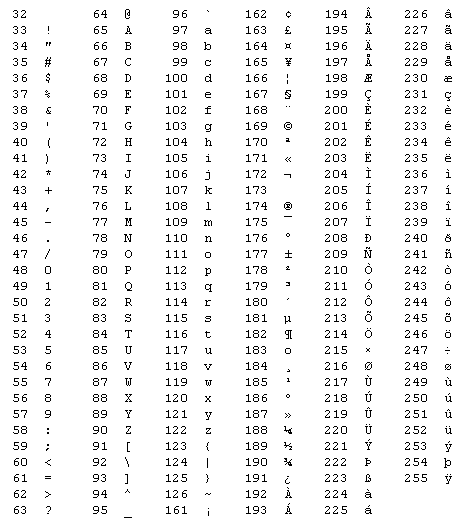
60 < 92 \ 124 | 190 ¾ 222 Þ 254 þ

61 = 93 ] 125 } 191 ¿ 223 ß 255 ÿ

62 > 94 ^ 126 ~ 192 À 224 à

63 ? 95 \_ 161 ¡ 193 Á 225 á

**21. Caracteres ASCII imprimíveis (ISO-8859-1) - imagem**



**22. Códigos prontos para copiar e colar**

|  |
| --- |
| **Condicionais com o IF** |
| if [ -f "$arquivo" ]; then echo 'Arquivo encontrado'; fi |
| if [ ! -d "$dir" ]; then echo 'Diretório não encontrado'; fi |
| if [ $i -gt 5 ]; then echo 'Maior que 5'; else echo 'Menor que 5'; fi |
| if [ $i -ge 5 -a $i -le 10 ]; then echo 'Entre 5 e 10, incluindo'; fi |
| if [ $i -eq 5 ]; then echo '=5'; elif [ $i -gt 5 ]; then echo '>5'; else echo '<5'; fi |
| if [ "$USER" = 'root' ]; then echo 'Oi root'; fi |
| if grep -qs 'root' /etc/passwd; then echo 'Usuário encontrado'; fi |
| **Condicionais com o E (&&) e OU (||)** |
| [ -f "$arquivo" ] && echo 'Arquivo encontrado' |
| [ -d "$dir" ] || echo 'Diretório não encontrado' |
| grep -qs 'root' /etc/passwd && echo 'Usuário encontrado' |
| cd "$dir" && rm "$arquivo" && touch "$arquivo" && echo 'feito!' |
| [ "$1" ] && param=$1 || param='valor padrão' |
| [ "$1" ] && param=${1:-valor padrão} |
| [ "$1" ] || { echo "Uso: $0 parâmetro" ; exit 1 ; } |
| **Adicionar 1 à variável $i** |
| i=$(expr $i + 1) |
| i=$((i+1)) |
| let i=i+1 |
| let i+=1 |
| let i++ |
| **Loop de 1 à 10** |
| for i in 1 2 3 4 5 6 7 8 9 10; do echo $i; done |
| for i in $(seq 10); do echo $i; done |
| for ((i=1;i<=10;i++)); do echo $i; done |
| i=1 ; while [ $i -le 10 ]; do echo $i ; i=$((i+1)) ; done |
| i=1 ; until [ $i -gt 10 ]; do echo $i ; i=$((i+1)) ; done |
| **Loop nas linhas de um arquivo ou saída de comando** |
| cat /etc/passwd | while read LINHA; do echo "$LINHA"; done |
| grep 'root' /etc/passwd | while read LINHA; do echo "$LINHA"; done |
| while read LINHA; do echo "$LINHA"; done < /etc/passwd |
| while read LINHA; do echo "$LINHA"; done < <(grep 'root' /etc/passwd) |
| **Curingas nos itens do comando case** |
| case "$dir" in /home/\*) echo 'dir dentro do /home';; esac |
| case "$user" in root|joao|maria) echo "Oi $user";; \*) echo "Não te conheço";; esac |
| case "$var" in ?) echo '1 letra';; ??) echo '2 letras';; ??\*) echo 'mais de 2';; esac |
| case "$i" in [0-9]) echo '1 dígito';; [0-9][0-9]) echo '2 dígitos';; esac |
| **Caixas do Dialog** |
| dialog --calendar 'abc' 0 0 31 12 1999 |
| dialog --checklist 'abc' 0 0 0 item1 'desc1' on item2 'desc2' off |
| dialog --fselect /tmp 0 0 |
| (echo 50; sleep 2; echo 100) | dialog --gauge 'abc' 8 40 0 |
| dialog --infobox 'abc' 0 0 |
| dialog --inputbox 'abc' 0 0 |
| dialog --passwordbox 'abc' 0 0 |
| dialog --menu 'abc' 0 0 0 item1 'desc1' item2 'desc2' |
| dialog --msgbox 'abc' 8 40 |
| dialog --radiolist 'abc' 0 0 0 item1 'desc1' on item2 'desc2' off |
| dialog --tailbox /tmp/arquivo.txt 0 0 |
| dialog --textbox /tmp/arquivo.txt 0 0 |
| dialog --timebox 'abc' 0 0 23 59 00 |
| dialog --yesno 'abc' 0 0 |
| **Dica1:** dialog ... && echo 'Apertou OK/Yes' || echo 'Apertou Cancel/No' |
| **Dica2:** resposta=$(dialog --stdout --TIPODACAIXA 'abc' ...) |

**23. Atalhos da linha de comando (set -o emacs)**

|  |  |  |
| --- | --- | --- |
| **Atalho** | **Descrição** | **Tecla Similar** |
| Ctrl+A | Move o cursor para o início da linha | Home |
| Ctrl+B | Move o cursor uma posição à esquerda | ← |
| Ctrl+C | Envia sinal EOF() para o sistema |  |
| Ctrl+D | Apaga um caractere à direita | Delete |
| Ctrl+E | Move o cursor para o fim da linha | End |
| Ctrl+F | Move o cursor uma posição à direita | → |
| Ctrl+H | Apaga um caractere à esquerda | Backspace |
| Ctrl+I | Completa arquivos e comandos | Tab |
| Ctrl+J | Quebra a linha | Enter |
| Ctrl+K | Recorta do cursor até o fim da linha |  |
| Ctrl+L | Limpa a tela (igual ao comando clear) |  |
| Ctrl+N | Próximo comando |  |
| Ctrl+P | Comando anterior |  |
| Ctrl+Q | Destrava a shell (veja Ctrl+S) |  |
| Ctrl+R | Procura no histórico de comandos |  |
| Ctrl+S | Trava a shell (veja Ctrl+Q) |  |
| Ctrl+T | Troca dois caracteres de lugar |  |
| Ctrl+U | Recorta a linha inteira |  |
| Ctrl+V | Insere caractere literal |  |
| Ctrl+W | Recorta a palavra à esquerda |  |
| Ctrl+X | Move o cursor para o início/fim da linha (2x) | Home/End |
| Ctrl+Y | Cola o trecho recortado |  |

**24. A caixa de ferramentas do shelleiro**

|  |  |  |
| --- | --- | --- |
| **Comando** | **Função** | **Opções úteis** |
| **cat** | *Mostra arquivo* | -n, -s |
| **cut** | *Extrai campo* | -d -f, -c |
| **date** | *Mostra data* | -d, +'...' |
| **diff** | *Compara arquivos* | -u, -Nr, -i, -w |
| **echo** | *Mostra texto* | -e, -n |
| **find** | *Encontra arquivos* | -name, -iname, -type f, -exec, -or |
| **fmt** | *Formata parágrafo* | -w, -u |
| **grep** | *Encontra texto* | -i, -v, -r, -qs, -n, -l, -w -x, -A -B -C |
| **head** | *Mostra início* | -n, -c |
| **od** | *Mostra caracteres* | -a, -c, -o, -x |
| **paste** | *Paraleliza arquivos* | -d, -s |
| **printf** | *Mostra texto* | *nenhuma* |
| **rev** | *Inverte texto* | *nenhuma* |
| **sed** | *Edita texto* | -n, -f, s/isso/aquilo/, p, d, q, N |
| **seq** | *Conta números* | -s, -f |
| **sort** | *Ordena texto* | -n, -f, -r, -k -t, -o |
| **tac** | *Inverte arquivo* | *nenhuma* |
| **tail** | *Mostra final* | -n, -c, -f |
| **tee** | *Arquiva fluxo* | -a |
| **tr** | *Transforma texto* | -d, -s, A-Z a-z |
| **uniq** | *Remove duplicatas* | -i, -d, -u |
| **wc** | *Conta letras* | -c, -w, -l, -L |
| **xargs** | *Gerencia argumentos* | -n, -i |

* Quer imprimir o canivete? [Baixe o PDF](http://aurelio.net/shell/canivete/pdf/).
* Veja também: [miniman](http://aurelio.net/shell/miniman/), o irmãozinho do canivete suíço.
* Veja o [código-fonte do canivete](http://aurelio.net/shell/canivete/index.t2t), no formato [txt2tags](http://txt2tags.org/pt/).
* Traduções do canivete: [Español](http://aurelio.net/shell/canivete/es/), [English](http://aurelio.net/shell/canivete/en/)

[[](http://aurelio.net/shell/)Aprenda mais sobre Shell Script](http://aurelio.net/shell/)

<http://www.devin.com.br/shell_script/>

# [Programando em shell script](http://www.devin.com.br/shell_script/)

[[Comentários](http://www.devin.com.br/shell_script/#comments)  190](http://www.devin.com.br/shell_script/#comments)

Visualizações

883,093

 1077 10

 22 104

Índice de Conteúdo

* [Primeira parte, uma introdução](http://www.devin.com.br/shell_script/#Primeira-parte-uma-introduo)
  + [Conceito de Variáveis em shell script](http://www.devin.com.br/shell_script/#Conceito-de-Variveis-em-shell-script)
  + [Controle de fluxo com o if](http://www.devin.com.br/shell_script/#Controle-de-fluxo-com-o-if)
  + [Outras alternativas](http://www.devin.com.br/shell_script/#Outras-alternativas)
* [Segunda parte, se aprofundando mais!](http://www.devin.com.br/shell_script/#Segunda-parte-se-aprofundando-mais)
  + [case](http://www.devin.com.br/shell_script/#case)
  + [for](http://www.devin.com.br/shell_script/#for)
  + [while](http://www.devin.com.br/shell_script/#while)
  + [until](http://www.devin.com.br/shell_script/#until)
  + [Usando vários scripts em um só](http://www.devin.com.br/shell_script/#Usando-vrios-scripts-em-um-s)
  + [Variáveis especiais](http://www.devin.com.br/shell_script/#Variveis-especiais)
  + [Funções](http://www.devin.com.br/shell_script/#Funes)
  + [Exemplo Final](http://www.devin.com.br/shell_script/#Exemplo-Final)
* [Terceira parte, janelas graficas](http://www.devin.com.br/shell_script/#Terceira-parte-janelas-graficas)

## Primeira parte, uma introdução

Quem usa Linux conhece bem o prompt de comando sh, ou variações como o bash. O ue muita gente não sabe é que o sh ou o bash têm uma “poderosa” linguagem de script embutido nelas mesmas – shell script. Diversas pessoas utilizam-se desta linguagem para facilitar a realização de inúmeras tarefas administrativas no Linux, ou até mesmo criar seus próprios programinhas. Patrick Volkerding, criador da distribuição Slackware, utiliza esta linguagem para toda a instalação e configuração de sua distribuição. Você poderá criar scripts para automar as tarefas diárias de um servidor, para efetuar backup automático regularmente, procurar textos, criar formatações, e muito mais. Para você ver como esta linguagem pode ser útil, vamos ver alguns passos introdutórios sobre ela.

Interpretadores de comandos são programas feitos para intermediar o usuário e seu sistema. Através destes interpretadores, o usuário manda um comando, e o interpretador o executa no sistema. Eles são a “Shell” do sistema Linux. Usaremos o interpretador de comandos bash, por ser mais “extenso” que o sh, e para que haja uma melhor compreensão das informações obtidas aqui, é bom ter uma base sobre o conceito de lógica de programação.

Uma das vantagens destes shell scripts é que eles não precisam ser compilados, ou seja, basta apenas criar um arquivo texto qualquer, e inserir comandos à ele. Para dar à este arquivo a definição de “shell script”, teremos que incluir uma linha no começo do arquivo (#!/bin/bash) e torná-lo “executável”, utilizando o comando chmod. Vamos seguir com um pequeno exemplo de um shell script que mostre na tela: “Nossa! Estou vivo!”:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2 | #!/bin/bash  echo 'Nossa! Estou vivo!' |

Fácil, hein? A primeira linha indica que todas as outras linhas abaixo deverão ser executadas pelo bash (que se localiza em /bin/bash), e a segunda linha imprimirá na tela a frase “Nossa! Estou vivo!”, utilizando o comando echo, que serve justamente para isto. Como você pôde ver, todos os comandos que você digita diretamente na linha de comando, você poderá incluir no seu shell script, criando uma série de comandos, e é essa combinação de comandos que forma o chamado shell script. Tente também dar o comando ‘file arquivo’ e veja que a definição dele é de Bourne-Again Shell Script (bash script).

Contudo, para o arquivo poder se executável, você tem de atribuir o comando de executável para ele. E como citamos anteriormente, o comando chmod se encarrega disto:

|  |
| --- |
| $ chmod +x arquivo |

Pronto, o arquivo poderá ser executado com um simples “./arquivo”.

### Conceito de Variáveis em shell script

Variáveis são caracteres que armazenam dados, uma espécie de atalho. O bash reconhece uma variável quando ela começa com $, ou seja, a diferença entre ‘palavra’ e ‘$palavra’ é que a primeira é uma palavra qualquer, e a outra uma variável. Para definir uma variável, utilizamos a seguinte sintaxe:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1 | variavel="valor" |

O ‘valor’ será atribuído a ‘variável ‘. Valor pode ser uma frase, números, e até outras variáveis e comandos. O valor pode ser expressado entre as aspas (“”), apóstrofos (”) ou crases (“). As aspas vão interpretar as variáveis que estiverem dentro do valor, os apóstrofos lerão o valor literalmente, sem interpretar nada, e as crases vão interpretar um comando e retornar a sua saída para a variável.

Vejamos exemplos:

|  |
| --- |
| $ variavel="Eu estou logado como usuário $user"  $ echo $variavel  Eu estou logado como usuário cla    $ variavel='Eu estou logado como usuário $user'  $ echo $variavel  Eu estou logado como usuário $user    $ variavel="Meu diretório atual é o `pwd`"  $ echo $variavel  Meu diretório atual é o /home/cla |

Se você quiser criar um script em que o usuário deve interagir com ele, é possível que você queira que o próprio usuário defina uma variável, e para isso usamos o comando read, que dará uma pausa no script e ficarará esperando o usuário digitar algum valor e teclar enter. Exemplo:

|  |
| --- |
| echo "Entre com o valor para a variável: " ; read variavel |

(O usuário digita e tecla enter, vamos supor que ele digitou ‘eu sou um frutinha’)

|  |
| --- |
| echo $variavel  eu sou um frutinha |

### Controle de fluxo com o if

Controle de fluxo são comandos que vão testando algumas alternativas, e de acordo com essas alternativas, vão executando comandos. Um dos comandos de controle de fluxo mais usados é certamente o if, que é baseado na lógica “se acontecer isso, irei fazer isso, se não, irei fazer aquilo”.

Vamos dar um exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if [ -e $linux ]  then    echo 'A variável $linux existe.'  else    echo 'A variável $linux não existe.'  fi |

O que este pedaço de código faz? O if testa a seguinte expressão: Se a variável $linux existir, então (then) ele diz que que existe com o echo, se não (else), ele diz que não existe. O operador -e que usei é pré-definido, e você pode encontrar a listagem dos operadores na tabela:

|  |  |
| --- | --- |
| -eq | Igual |
| -ne | Diferente |
| -gt | Maior |
| -lt | Menor |
| -o | Ou |
| -d | Se for um diretório |
| -e | Se existir |
| -z | Se estiver vazio |
| -f | Se conter texto |
| -o | Se o usuário for o dono |
| -r | Se o arquivo pode ser lido |
| -w | Se o arquivo pode ser alterado |
| -x | Se o arquivo pode ser executado |

### Outras alternativas

Existem inúmeros comandos no Linux, e para explicar todos, teríamos de publicar um verdadeiro livro. Mas existem outras possibilidades de aprendizado desta língua, que também é usado em todas as programações. Primeiro de tudo você pode dar uma olhada na manpage do bash (comando man bash), que disponibilizará os comandos embutidos no interpretador de comandos. Uma das coisas essencias para o aprendizado é sair coletando exemplos de outros scripts e ir estudando-os minuciosamente. Procure sempre comandos e expressões novas em outros scripts e em manpages dos comandos. E por último, mas não o menos importante, praticar bastante!

Na tabela a seguir, você pode encontrar uma listagem de comandos para usar em seu shell script:

|  |  |
| --- | --- |
| echo | Imprime texto na tela |
| read | Captura dados do usuário e coloca numa variável |
| exit | Finaliza o script |
| sleep | Dá uma pausa em segundos no script |
| clear | Limpa a tela |
| stty | Configura o terminal temporariamente |
| tput | Altera o modo de exibição |
| if | Controle de fluxo que testa uma ou mais expressões |
| case | Controle de fluxo que testa várias expressões ao mesmo tempo |
| for | Controle de fluxo que testa uma ou mais expressões |
| while | Controle de fluxo que testa uma ou mais expressões |

E assim seja, crie seus próprios scripts e facilite de uma vez só parte de sua vida no Linux!

## Segunda parte, se aprofundando mais!

Falamos sobre o conceito da programação em Shell Script, e demos o primeiro passo para construir nossos próprios scripts. Agora vamos nos aprofundar nos comandos mais complicados, aprendendo a fazer programas ainda mais úteis. Nestes comandos estão inclusos o case e os laços for, while e until. Além disso, vamos falar de funções e, por último, teremos um programa em shell script.

### case

O case é para controle de fluxo, tal como é o if. Mas enquanto o if testa expressões não exatas, o case vai agir de acordo com os resultados exatos. Vejamos um exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | case $1 in    parametro1) comando1 ; comando2 ;;    parametro2) comando3 ; comando4 ;;    \*) echo "Você tem de entrar com um parâmetro válido" ;;  esac |

Aqui aconteceu o seguinte: o case leu a variável $1 (que é o primeiro parâmetro passado para o programa), e comparou com valores exatos. Se a variável $1 for igual à “parametro1″, então o programa executará o comando1 e o comando2; se for igual à “parametro2″, executará o comando3 e o comando4, e assim em diante. A última opção (\*), é uma opção padrão do case, ou seja, se o parâmetro passado não for igual a nenhuma das outras opções anteriores, esse comando será executado automaticamente.

Você pode ver que, com o case fica muito mais fácil criar uma espécie de “menu” para o shell script do que com o if. Vamos demonstrar a mesma função anterior, mas agora usando o if:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | if [ -z $1 ]; then    echo "Você tem de entrar com um parâmetro válido"    exit  elif [ $1 = "parametro1" ]; then    comando1    comando2  elif [ $1 = "parametro2" ]; then    comando3    comando4  else    echo "Você tem de entrar com um parâmetro válido"  fi |

Veja a diferença. É muito mais prático usar o case! A vantagem do if é que ele pode testar várias expressões que o case não pode. O case é mais prático, mas o if pode substituí-lo e ainda abrange mais funções. Note que, no exemplo com o if, citamos um “comando” não visto antes: o elif – que é uma combinação de else e if. Ao invés de fechar o if para criar outro, usamos o elif para testar uma expressão no mesmo comando if.

### for

O laço for vai substituindo uma variável por um valor, e vai executando os comandos pedidos. Veja o exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | for i in \*  do    cp $i $i.backup    mv $i.backup /usr/backup  done |

Primeiramente o laço for atribuiu o valor de retorno do comando “\*” (que é equivalente a um ls sem nenhum parâmetro) para a variável $i, depois executou o bloco de comandos. Em seguida ele atribui outro valor do comando “\*” para a variável $1 e reexecutou os comandos. Isso se repete até que não sobrem valores de retorno do comando “\*”. Outro exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for original in \*; do    resultado=`echo $original |               tr '[:upper:]' '[:lower:]'`      if [ ! -e $resultado ]; then      mv $original $resultado    fi  done |

Aqui, o que ocorre é a transformação de letras maiúsculas para minúsculas. Para cada arquivo que o laço lê, uma variável chamada $resultado irá conter o arquivo em letras minúsculas. Para transformar em letras minúsculas, usei o comando tr. Caso não exista um arquivo igual e com letras minúsculas, o arquivo é renomeado para o valor da variável $resultado, de mesmo nome, mas com letras minúsculas.

Como os exemplos ilustram, o laço for pode ser bem útil no tratamento de múltiplos arquivos. Você pode deixá-los todos com letras minúsculas ou maiúsculas sem precisar renomear cada um manualmente, pode organizar dados, fazer backup, entre outras coisas.

### while

O while testa continuamente uma expressão, até que ela se torne falsa. Exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | variavel="valor"  while [ $variavel = "valor" ]; do    comando1    comando2  done |

O que acontece aqui é o seguinte: enquanto a “$variavel” for igual a “valor”, o while ficará executando os comandos 1 e 2, até que a “$variavel” não seja mais igual a “valor”. Se no bloco dos comandos a “$variavel” mudasse, o while iria parar de executar os comandos quando chegasse em done, pois agora a expressão $variavel = “valor” não seria mais verdadeira.

### until

Tem as mesmas características do while, a única diferença é que ele faz o contrário. Veja o exemplo abaixo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | variavel="naovalor"  until [ $variavel = "valor" ]; do    comando1    comando2  done |

Ao invés de executar o bloco de comandos (comando1 e comando2) até que a expressão se torne falsa, o until testa a expressão e executa o bloco de comandos até que a expressão se torne verdadeira. No exemplo, o bloco de comandos será executado desde que a expressão $variavel = “valor” não seja verdadeira. Se no bloco de comandos a variável for definida como “valor”, o until pára de executar os comandos quando chega ao done.

Vejamos um exemplo para o until que, sintaticamente invertido, serve para o while também:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | var=1  count=0  until [ $var = "0" ]; do    comando1    comando2    if [ $count = 9 ]; then      var=0    fi    count=`expr $count + 1`  done |

Primeiro, atribuímos à variável “$var” o valor “1″. A variável “$count” será uma contagem para quantas vezes quisermos executar o bloco de comandos. O until executa os comandos 1 e 2, enquanto a variável “$var” for igual a “0″. Então usamos um if para atribuir o valor 0 para a variável “$var”, se a variável “$count” for igual a 9. Se a variável “$count” não for igual a 0, soma-se 1 a ela. Isso cria um laço que executa o comando 10 vezes, porque cada vez que o comando do bloco de comandos é executado, soma-se 1 à variável “$count”, e quando chega em 9, a variável “$var” é igualada a zero, quebrando assim o laço until.

### Usando vários scripts em um só

Pode-se precisar criar vários scripts shell que fazem funções diferentes, mas, e se você precisar executar em um script shell um outro script externo para que este faça alguma função e não precisar reescrever todo o código? É simples, você só precisa incluir o seguinte comando no seu script shell:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1 | . bashscript2 |

Isso executará o script shell “bashscript2″ durante a execução do seu script shell. Neste caso ele será executado na mesma script shell em que está sendo usado o comando. Para utilizar outra shell, você simplesmente substitui o “.” pelo executável da shell, assim:

|  |
| --- |
| sh script2  tcsh script3 |

Nessas linhas o script2 será executado com a shell sh, e o script3 com a shell tcsh.

### Variáveis especiais

|  |  |
| --- | --- |
| $0 | Nome do script que está sendo executado |
| $1-$9 | Parâmetros passados à linha de comando |
| $# | Número de parâmetros passados |
| $? | Valor de retorno do último comando ou de todo o shell script. (o comando “exit 1″ retorna o valor 1) |
| $$ | Número do PID (Process ID) |

Você também encontra muitas variáveis, já predefinidas, na página de manual do bash (comando “man bash”, seção Shell Variables).

### Funções

Funções são blocos de comandos que podem ser definidos para uso posterior em qualquer parte do código. Praticamente todas as linguagens usam funções que ajudam a organizar o código. Vejamos a sintaxe de uma função:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | funcao() {    comando1    comando2    [...]  } |

Fácil de entender, não? A função funcionará como um simples comando próprio. Você executa a função em qualquer lugar do script shell, e os comandos 1, 2 e outros serão executados. A flexibilidade das funções permite facilitar a vida do programador, como no exemplo final.

### Exemplo Final

Agora vamos dar um exemplo de um programa que utilize o que aprendemos com os artigos.

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | #!/bin/bash  # Exemplo Final de Script Shell  #    Principal() {    echo "Exemplo Final sobre o uso de scripts shell"    echo "------------------------------------------"    echo "Opções:"    echo    echo "1. Trasformar nomes de arquivos"    echo "2. Adicionar um usuário no sistema"    echo "3. Deletar um usuário no sistema"    echo "4. Fazer backup dos arquivos do /etc"    echo "5. Sair do exemplo"    echo    echo -n "Qual a opção desejada? "    read opcao    case $opcao in      1) Transformar ;;      2) Adicionar ;;      3) Deletar ;;      4) Backup ;;      5) exit ;;      \*) "Opção desconhecida." ; echo ; Principal ;;    esac  }    Transformar() {    echo -n "Para Maiúsculo ou minúsculo? [M/m] "    read var    if [ $var = "M" ]; then      echo -n "Que diretório? "      read dir        for x in `/bin/ls` $dir; do        y=`echo $x | tr '[:lower:]' '[:upper:]'`        if [ ! -e $y ]; then          mv $x $y        fi      done      elif [ $var = "m" ]; then      echo -n "Que diretório? "      read dir        for x in `/bin/ls` $dir; do        y=`echo $x | tr '[:upper:]' '[:lower:]'`        if [ ! -e $y ]; then          mv $x $y        fi      done      fi  }    Adicionar() {    clear    echo -n "Qual o nome do usuário a se adicionar? "    read nome    adduser nome    Principal  }    Deletar() {    clear    echo -n "Qual o nome do usuário a deletar? "    read nome    userdel nome    Principal  }    Backup() {    for x in `/bin/ls` /etc; do      cp -R /etc/$x /etc/$x.bck      mv /etc/$x.bck /usr/backup    done  }    Principal |

## Terceira parte, janelas graficas

Nos dois topicos anteriores, vimos algumas coisas básicas e lógicas de programação em shell no Linux. Agora para completar, darei aqui dicas de como usar janelas gráficas em seus shell scripts. Isso mesmo, janelas que rodam no ambiente gráfico, utilizadas facilmente em seu shell script. Com esse recurso, vai ser possível deixar o seu programa bem amigável.

Não vai ser preciso saber muita coisa de programação em shell, pois é muito simples. Através do programa Xdialog, você poderá criar os mais variados tipos de janelas para o programa: caixas de texto, seleçÕes, radios, etc. O Xdialog é uma idéia que vem do dialog/cdialog, um programa para console que gera “janelas” no console (aquelas da instalação do Slackware) usando a biblioteca ncurses. O Xdialog ao invés de usar a biblioteca ncurses, usa a Xlib para criar as janelas no ambiente gráfico.

Primeiro de tudo será necessário você obter o Xdialog no seu sistema. Não é comum o Xdialog estar incluso nas distribuições, então você terá de pegar e compilar o programa. Obtenha o programa no CD da Revista ou visite o endereço oficial do Xdialog, que é http://xdialog.free.fr. Aqui eu peguei o arquivo Xdialog-1.4.5.tar.bz2, e agora vamos aos passos básicos para instalar ele. Primeiro descompacte-o com o comando bunzip2 Xdialog-1.4.5.tar.bz2, e logo em seguida tar xpvf Xdialog-1.4.5.tar. Um diretório chamado Xdialog-1.4.5 será criado, e entrando nele você estará pronto para compilá-lo e instalá-lo. Para fazer isso use os comandos ./configure, depois make e por último make install. No passo do make install, o binário do Xdialog será colocado em /usr/local/bin. Pronto, agora você já poderá utilizar o Xdialog através de qualquer shell script.

E agora vamos à ação! Como aprendemos nos artigos anteriores, em shell script é só colocar o comando dentro do arquivo que ele já vai ser executado quando o script for executado. Então só o que temos de aprender aqui é como usar o Xdialog. Vamos ver um primeiro exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #!/bin/bash    Xdialog --title "Exemplo número 1!" --center --stdout --yesno \  "Isto é legal?" \  0 0    echo $? |

Como você pôde ver, o programa Xdialog gerou uma janela com título “Exemplo número 1!”, perguntando “Isto é legal?” e com opções de Sim e Não. Note que a \ (barra) serve para indicar à shell para continuar o comando da linha seguinte, então estas três linhas são que nem uma só. Como último comando do exemplo dado, temos o echo $?, que eu coloquei apenas para indicar qual foi o retorno da pergunta. Caso o usuário apertou em Sim, o retorno vai ser 0, e se apertou em Não, vai ser 1. Podemos usar este retorno para controlar o que o usuário escolher. Vejamos um exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Xdialog --title "Exemplo número 2!" --center --stdout --yesno \  "Neste exemplo, vamos ver o que você quer fazer. Você deseja continuar com o programa?" \  0 0    if [ $? = "0" ]; then    echo "Que bom! Você continuou o programa! Parabéns!"  elif [ $? = "1" ]; then    echo "Você saiu do programa..."  fi |

Viu como funciona? Agora vamos ver outros recursos que o Xdialog pode oferecer. Eu vou dar vários exemplos aqui e sair comentando cada opção. Você precisará praticar bastante e conhecer as várias opções. Primeiro vamos gerar uma simples mensagem pro usuário ver:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Xdialog --title "Aviso" --center --stdout --msgbox \  "Este programa é apenas um exemplo para você ver como o Xdialog \  \nfunciona. Apropósito, se você praticar bastante pode criar \  \nprogra mas incríveis e facilmente, que daria muito mais \  \ntrabalho fazendo em outras linguagens." \  0 0 |

O usuário aperta Ok e o shell script continua normalmente. No primeiro exemplo eu usei a opção –yesno que gerava o sim e não. Agora usei o –msgbox. Mas e se você quiser que o usuário digite algo e isto seja gravado em um arquivo por exemplo? Vamos ver este exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4 | Xdialog --title "Que Anime que você mais gosta?" --center --inputbox \  "Se você sabe o que é Anime, e gosta, qual o seu preferido?\n \   Sua resposta será gravada no arquivo resposta." \  0 0 2 > resposta |

Depois que o usuário preenche o campo e dá Ok, a resposta que este usuário digitou será gravada no arquvio resposta. Isto ocorreu pelo fato de eu ter colocado o direcionador 2> para o arquivo resposta. Se eu colocasse a opção –stdout na linha de comando do Xdialog, a resposta do usuário apareceria na tela. Tente você.

Vamos ver agora seleção de itens, que é uma das coisas mais importantes num programa. Desta vez usaremos a opção –menubox para gerar um menu com os itens a serem selecionados. Mais uma vez, vamos ao exemplo:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Xdialog --title "Exemplo de Menu" --center --stdout --menubox \  "Qual sua distribuição Linux favorita?" \  20 50 0 \  1 "Slackware" \  2 "Debian" \  3 "Red Hat" \  4 "Conectiva Linux" \  5 "Eu tenho minha própria distribuição" |

Viu como é fácil? O que o usuário escolher vai aparecer como resultado no console (por causa da opção –stdout). Se eu colocasse o redirecionador 2>, poderia ir para um arquivo como no exemplo anterior. Vamos esclarecer uma coisa aqui também… Depois do texto “Qual sua distribuição Linux favorita?”, há 2 números. Estes dois números correspondem à altura e comprimento da janela. Nos exemplos anteriores eu coloquei “0 0″ pois aí o Xdialog dimensiona automaticamente o tamanho da janela. Então já sabe, se quiser mudar o tamanho da janela, é só mudar estes dois números.

Agora como último exemplo, vamos criar uma janela em que o usuário poderá escolher uma ou mais opções. Isto é chamado de checklist, e pode ser visto no exemplo a seguir:

[?](http://www.devin.com.br/shell_script/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Xdialog --title "Último exemplo - checklist" --center --checklist \  "Como se pronuncia Linux?" \  0 0 0 \  "Opção 1" "Láinucs" off \  "Opção 2" "Lenocs" off \  "Opção 3" "Linúcs" off \  "Opçào 4" "Línucs" on \  "Opção 5" "GNUUU/Linux" off |

Veja agora a diferença entre esta checklist e o menu do exemplo anterior. Verifique que depois de cada opção há o on e o off. O on indica que esta opção deverá já estar marcada, e o off que não deverá estar marcada. Se o usuário escolher 3 opções, as mesmas serão o resultado.

Bem fácil criar uma interface amigável para suas shell scripts. Se você pensa que isto é tudo, ainda tem muito mais por trás do Xdialog. Para não ter que ficar comentando cada opção, vou dar uma lista de parâmetros e a descrição de suas funções. O que você deverá fazer é sair testando todas as opções e se impressionar :)

|  |  |
| --- | --- |
| –yesno | Uma janela com opções de “Sim” ou “Não” |
| –msgbox | Apenas mostra um texto informativo |
| –infobox | Mesmo que –msgbox, só que desaparece automaticamente em um determinado tempo |
| –inputbox | O usuário preenche um campo |
| –rangebox | Escolhe um número entre X e Y, com uma barra de rolagem |
| –textbox | Mostra o conteúdo de um arquivo numa caixa de texto |
| –editbox | Edita o conteúdo de um arquivo numa caixa de texto |
| –menubox | Cria um Menu de opções, onde se seleciona um item |
| –checklist | Mesmo que –menubox, só que pode-se selecionar vários itens |
| –radiolist | Mesmo que –menubox, mas agora em outro estilo |
| –treeview | Opções organizadas em forma de “árvore” (interessante) |
| –gauge | Um indicador de processo |
| –tailbox | Mostra o conteúdo de um arquivo |
| –fselect | Abre uma janela de seleção de um arquivo |
| –dselect | Abre uma janela de seleção de um diretório |
| –calendar | Mostra um calendário para escolher uma data |
| –timebox | Mostra uma janela para edição de horário |

Você também pode encontrar alguns exemplos no diretório samples que vem junto com o código-fonte do programa. Se você fizer algum programa legal em shell script, sinta-se a vontade para me mandar um e-mail. Outra coisa, como o Xdialog é uma idéia tirada do dialog/cdialog (existe também o whiptail que é parecido), você pode usar a mesma sintaxe para criar estas “janelas” no modo console. Espero que estas informações foram úteis a você e até a próxima :)

<http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/>

# [Shell Script: tratamento de argumentos e opções](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

[[Comentários](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/#comments)  10](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/#comments)

Visualizações

304,683

 36 4

 3 3

Na grande maioria das vezes, a linguagem shell script é usada para criar uma sequência de comandos que automatizam uma tarefa. Nisso, ela é extremamente eficiente e rápida. Combinar comandos é uma grande vantagem que o jeito UNIX de ser nos trouxe: várias ferramentas que fazem tarefas básicas, especializadas, e que quando se juntam realizam grandes feitos trabalhando juntas. Mas isso nunca impediu que se criasse também programas completos em shell script.

Uma característica para deixar um shell-script mais robusto e menos “sequencial/batch-mode” é o tratamento de argumentos. No meu clássico tutorial [Programando em Shell-Script](http://www.devin.com.br/shell_script/), o tópico Variáveis Especiais nos traz os primeiros itens que devemos aprender para o tratamento de argumentos. Existem variáveis especiais que tratam os argumentos passados para um programa ou uma função. Estes são:

* **$0** – Retorna o nome do script que foi executado
* **$N** – Onde N é um número, corresponde ao argumento passado (1 = primeiro argumento, 2 = segundo argumento, 3 = terceiro argumento, etc)
* **$\*** – Retorna todos os argumentos de uma vez.
* **$#** – Retorna a quantidade de argumentos passado para o script. (argc)

Vejamos agora um shell-script exempo que faz uso de todos esses argumentos:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | #!/bin/bash    if [ $# -lt 1 ]; then     echo "Faltou utilizar pelo menos um argumento!"     exit 1  fi    echo "Numero de argumentos: $#"    COUNT=0  for ARG in $\*; do     COUNT=`expr $COUNT + 1`     echo "Argumento $COUNT: $ARG"  done |

As linhas 3 a 6 verificam se a quantidade de argumentos ($#) é menor (-lt – less than) que 1. Ou seja, se o usuário não chamou o programa com nenhum argumento, ele imprime um erro e sai do programa com status 1.

A linha 8 mostra quantos argumentos foram utilizados, usando novamente o $#.

O resto das linhas, 10 a 14, usam o $\* com um laço for e um contador para mostrar quais foram os argumentos.

Executando agora este script sem argumentos:

|  |
| --- |
| $ ./tmp.sh  Faltou utilizar pelo menos um argumento! |

Agora executando com dois argumentos:

|  |
| --- |
| $ ./tmp.sh naosei testando  Numero de argumentos: 2  Argumento 1: naosei  Argumento 2: testando |

E agora com 4 argumentos:

|  |
| --- |
| $ ./tmp.sh a b c d  Numero de argumentos: 4  Argumento 1: a  Argumento 2: b  Argumento 3: c  Argumento 4: d |

Bem simples né?

## Argumentos como opções e seus valores

Algo comum que vemos nos programas são opções. Opções não deixam de ser argumentos para um programa, mas eles tem um significado especial. Do tipo: Se a opção -d existir, ativar durante o programa o modo de depuração. Se houver um -h, então mostre uma ajuda e não faça mais nada. Se houver um -v mostre a versão, e por aí vai.

Exemplo:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | #!/bin/bash    case $1 in     "-h") echo "Isto seria uma ajuda... Mas fiquei com preguiça de escrevê-la."           ;;     "-v") echo "Versão 666."           ;;     \*) echo "Opção inválida!"        exit 1        ;;  esac |

Exemplos do uso do script:

|  |
| --- |
| $ ./tmp.sh -h  Isto seria uma ajuda... Mas fiquei com preguiça de escrevela.    $ ./tmp.sh -v  Versão 666.    $ ./tmp.sh -O  Opção inválida!    $ ./tmp.sh  Opcao invalida! |

Com isso a gente resolve um problema e cria mais outros dois…

* E se o usuário colocar as duas opções? Só uma funcionaria.
* E se uma das opções precisasse de um valor? Estilo “-f arquivo.log” gravaria um arquivo de log com as operações.

Poderíamos escrever vários algoritmos que verificassem cada um de todos os argumentos, testasse se fosse um ou outro, utilizasse as opções… Mas felizmente não precisamos fazer nada disso! O bash conta com uma função interna que trata os argumentos: o famoso **getopts**.

## Utilizando o getopts para tratar tratar argumentos e opções

Seguindo a mesma linha de raciocínio, vamos logo para um exemplo de programa. Supondo que queiramos um shell-script que faça isso:

* Caso a opção -h seja usada, mostra a ajuda e sai do programa.
* Caso a opção -v seja usada, mostra a versão e sai do programa.
* Caso a opção -o seja usada, grava um arquivo de log com as operações efetuadas e resultados.
* Caso a opção -u seja usada, mostra o resultado do comando “uname -a”
* Caso a opção -m seja usada, mostra o resultado do comando “free -m”
* Caso a opção -s seja usada, mostra o resultado do comando “swap -s”

Note que apenas as opções -h e -v saem do programa após a execução. Agora vamos ao código:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | #!/bin/bash    function PrintUsage() {     echo "Uso: `basename $0` <-umsf> [-ohv]"     exit 1  }    while getopts "hvo:umsf" OPTION  do     case $OPTION in        h) PrintUsage           ;;        v) echo "`basename $0` versao 666."           exit           ;;        o) ARQUIVO\_LOG=$OPTARG           ;;        u) DO\_UNAME=1           ;;        m) DO\_FREE=1           ;;        s) DO\_SWAPON=1           ;;        ?) PrintUsage           ;;     esac  done  shift $((OPTIND-1))    if [ -z "$DO\_UNAME" ] && [ -z "$DO\_FREE" ] && [ -z "$DO\_SWAPON" ] && [ -z "$DO\_FDISK" ]; then     PrintUsage  fi    if [ "$ARQUIVO\_LOG" ]; then   echo "Execucao iniciada em `date`." >> $ARQUIVO\_LOG       if [ "$DO\_UNAME" == 1 ]; then        uname -a >> $ARQUIVO\_LOG     fi       if [ "$DO\_FREE" == 1 ]; then        free -m >> $ARQUIVO\_LOG     fi       if [ "$DO\_SWAPON" == 1 ]; then        swapon -s >> $ARQUIVO\_LOG     fi  else     echo "Execucao iniciada em `date`."     if [ "$DO\_UNAME" == 1 ]; then        uname -a     fi       if [ "$DO\_FREE" == 1 ]; then        free -m     fi       if [ "$DO\_SWAPON" == 1 ]; then        swapon -s     fi  fi |

O interessante para nós são as linhas 8 a 28. O laço while getopts começa a tratar todos os argumentos. A cada iteração do laço, ele coloca a letra da opção na variável ***$OPTION***.

Note que para cada opção que precisamos, colocamos uma letra no primeiro argumento do getopts:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 1 | while getopts "hvo:umsf" OPTION |

Note também que depois da letra o temos um dois pontos (:). Esse dois pontos significa que logo após a opção -o, o usuário precisa fornecer um valor. Este valor é automaticamente armazenado na variável ***$OPTARG***.

Dessa maneira, podemos executar esse programa de diversas formas:

|  |
| --- |
| ./tmp.sh -o arquivo.log -u  (executa o "uname -a" e grava no arquivo arquivo.log)    ./tmp.sh -um  (executa os comandos "uname -a" e "free -m")    ./tmp.sh -m -s -u  (executa os comandos "free -m", "swapon -s" e "uname -a") |

Ou seja, não importa a ordem, o getopts vai reconhecer e executar as ações de acordo com a opção especificada.

E se você colocar uma opção que não está contemplatada… O “?” do case irá ser executado, por exemplo:

|  |
| --- |
| $ ./tmp.sh -a  ./tmp.sh: illegal option -- a  Uso: tmp.sh <-umsf> [-ohv] |

E dessa forma fica bem fácil de entender e usar o getopts :) Depois que o laço é todo feito e executado em todos os argumentos (no meu caso, preferi apenas configurar variáveis para cada opção e tratá-las depois), ele executa o comando que está na linha 28:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 1 | shift $((OPTIND-1)) |

Este comando faz com que os **argumentos de opções** sejam “comidos“, até que não sobre nenhuma opção. Em outras palavras, os argumentos representados pelas variáveis **$N** só serão aqueles que não pertençam a nenhuma opção. Exemplo:

|  |
| --- |
| ./tmp.sh -u -o arquivo.log -m argumento1 argumento2 |

Nesse caso, o $1 seria o argumento1 e o $2 seria o argumento2, quando na verdade, sem o shift, eles seriam respectivamente o $5 e $6.

Como nem tudo é perfeito, a função getopts do bash **não aceita opções longas** (–nome-da-opcao), ou seja, voce só pode utilizar uma letra como opção. Represente bem suas opções com as letras! :)

## Argumentos dentro de funções

Se dentro de um shell-script temos uma função, essa função é enxergada pela shell como se fosse um comando. Nesse sentido, dentro de uma função as variáveis $N definidas pelo programa **não funcionarão**. Exemplo:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13 | #!/bin/bash    function Dummy() {     echo "Numero de argumentos: $#"       COUNT=0     for ARG in $\*; do        COUNT=`expr $COUNT + 1`        echo "Argumento $COUNT: $ARG"     done  }    Dummy |

Não importa o que você executar com o script acima, a saída será sempre a mesma: 0 números de argumentos, como mostrado a seguir.

|  |
| --- |
| $ ./tmp.sh  Numero de argumentos: 0    $ ./tmp.sh naosei temporario  Numero de argumentos: 0    $ ./tmp.sh a b c d e f g  Numero de argumentos: 0 |

Para a função **Dummy**, as variáveis especiais dos argumentos **funcionam apenas para a função** e não para o programa inteiro. É como se as variáveis fossem locais, e não globais. Vamos então substituir a linha da chamada da função Dummy (linha 13) por:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 1 | Dummy a b c d |

E tentar executar novamente:

|  |
| --- |
| $ ./tmp.sh  Numero de argumentos: 4  Argumento 1: a  Argumento 2: b  Argumento 3: c  Argumento 4: d |

Sabendo disso, não se percam na hora de usar os argumentos dentro das funções e lembrem-se que isto pode ser útil na hora de implementar diversas funções dentro de um script. Um bom exemplo disso é implementar a função PrintUsage que usamos anteriormente para, além de mostrar uma mensagem de uso, mostrar também uma mensagem de erro personalizada:

[?](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | function PrintUsage() {     [ "$1" ] && echo -ne "Erro: $1\n"     echo "Uso: $(basename $0)  <-umsf> [-ohv]"     exit 1  } |

Agora é so chamar a função como…

|  |
| --- |
| PrintUsage "Faltando parâmetros."  PrintUsage "Opção inválida."  PrintUsage "No donut for you." |

Use a criatividade de um programador (afinal, programação é arte) e comece a aprimorar suas ferramentas bash! :)

[[Comentários](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/#comments)  10](http://www.devin.com.br/shell-script-tratamento-de-argumentos-e-opcoes/#comments)

Visualizações

304,683

 36 4

 3 3

TagsLeia também

[argumentos](http://www.devin.com.br/tag/argumentos/) [opções](http://www.devin.com.br/tag/opcoes/) [shell](http://www.devin.com.br/tag/shell/) [shell-script](http://www.devin.com.br/tag/shell-script/) [tutorial](http://www.devin.com.br/tag/tutorial/)



### [Hugo Cisneiros (Eitch)](http://www.devin.com.br/author/admin/)

Apaixonado por Linux e administração de sistemas. Viciado em Internet, servidores, e em passar conhecimento. Idealizador do Devin, tem como meta aprender e ensinar muito Linux, o que ele vem fazendo desde 1997 :-)

O comando shift é um "built-in" do bash. Ele serve para "remover argumentos do começo", fazendo com que o próximo argumento "caiam na lista", por exemplo, você passou 3 argumentos, sendo eles **$1**, **$2** e **$3**, ao usar **shift**, ele "removerá" o argumento **$1**, e o argumento **$2** e **$3** "irão cair na lista", fazendo com que o argumento **$2** passe a ser o **$1** e o **$3** o **$2** respectivamente.  
  
Exemplificando, criarei o seguinte arquivo (**teste.sh**):

shift

echo $1

echo $2

Ao executar:

chmod +x teste.sh // Apenas para conseguir executar

./teste.sh arg1 arg2

arg2

Você poderá também quantizar os parâmetros que serão removidos (shift n):

shift 2

Dê uma olhada: <http://ss64.com/bash/shift.html>