O Shell

Fabricio Ferrari
fabricio@ferrari.pro.br

1 O que é o *Shell*

O *shell* é a ligação entre o usuário e o sistema. É ele quem interpreta os comandos entrados para outros aplicativos ou diretamente em chamadas de sistema. Além disso, os recursos do *shell* são indispensáveis para lidar com muitos arquivos ao mesmo tempo, para realizar uma tarefa repetidamente ou para programar uma ação para determinada ocasião, entre outros recursos. Começamos apresentando os tipos de *shell* mais difundidos e depois definindo alguns conceitos que serão úteis na sua utilização, para então tratarmos de exemplos práticos.

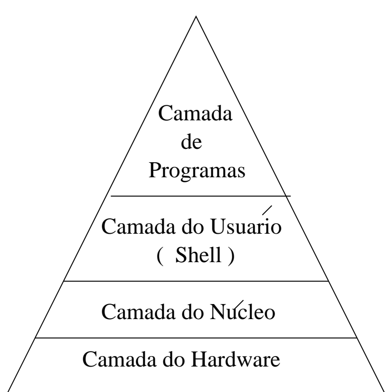

Figura 1: Estrutura do Unix.

2 Tipos de *Shell*

O mais comum de todos é o *shell* Bourne (`sh`), desenvolvido a partir de *shell* original, escrito em 1975 por S. R. Bourne. Mais tarde, alguns estudantes da Universidade da Califórnia em Berkeley criaram o *shell* C (`csh`), util para quem é familiar com a sintaxe da linguagem de programação C. O terceiro dos mais difundidos é o *shell* Korn, criado por Dadid Korn da AT&T, que preserva a funcionalidade do Bourne incluindo as características poderosas do *shell* C. Atualmente o Linux e a maioria dos sistemas UNIX contam com o `bash` ou Bourne

Again *shell*[1] e uma versão melhorada do *shell* C, o `tcsh`. Nossa discussão aqui está direcionada ao `tcsh`, mas os tópicos aplicam-se aos outros e as diferenças relevantes serão mencionadas. A escolha do *shell* depende principalmente das preferências pessoais de cada usuário, pois os principais recursos estão presentes em qualquer das alternativas.

3 Inicialização do *shell*

Para que o usuário abra sua sessão, tanto local como remotamente, é preciso que ele digite o seu nome e senha para o programa `login`, quem irá iniciar a sessão do usuário no sistema. O `login` examina se o nome informado consta na lista de usuários e se sua senha confere com aquela armazenada. Em caso de sucesso, o `login` chama o *shell* configurado para esse usuário. Esta informação está guardada no arquivo `/etc/passwd` na maioria dos sistemas[2], juntamente com o nome do usuário, sua senha criptografada, seu número de identificação, o número de identificação do seu grupo, uma descrição do usuário, seu diretório padrão (`HOME`) e o *shell* de seu uso, nesta ordem. Os campos estão separados por dois pontos (`:`).

Passados os processos de *login* em si, o *shell* executa os comandos nos arquivos de sistema `/etc/csh.cshrc` e `/etc/csh.login` e depois procura pelos arquivo de inicialização `.tcshrc`, no diretório `HOME` do usuário. Em caso de não existir este arquivo, o *shell* procura por `~/.cshrc`. finalmente o *shell* executa `~/.login` e `~/.history`, recaindo no *prompt*, quando está pronto para receber as entradas do usuário.

Os arquivos de inicialização em `/etc` servem para estabelecer configurações pertinentes às particularidades do sistema como um todo, enquanto os arquivos de inicialização de cada usuário permitem que cada um ajuste o comportamento do *shell* ao seu gosto. Estes arquivos podem contem quaisquer comandos que façam sentido ao *shell*, que os executará na ordem que os encontrar. As tarefas mais comuns realizadas a partir destes arquivos são gerenciamento básico de arquivos, como apagar arquivos temporários, definição de variáveis de ambiente (seção 10) de aplicativos e do próprio usuário e criação de apelidos de comandos (*alias*, seção 12).

[1]Um trocadilho em inglês que pode significar "o *shell* Bourne novamente" como também "o *shell* nascido de novo".

[2]Há situações em que essas informações estão escondidas em outro arquivo (*shadow passwords*) ou em que um servidor informa tais parâmetros para todas os clientes de uma rede (NIS). O primeiro caso ocorre por motivos de segurança e o segundo para manter a base de dados de usuários e senhas comum e sincronizada.

Figura 2: Um comando do *shell* como um filtro.

4 Descritores Padrão de Arquivos

Os arquivos no ambiente UNIX possuem forma livre, consistindo apenas de uma sequência de caracteres. As quebras de linha (nova linha) são delimitadas por caracteres de nova linha `\n`, enquanto o final de arquivo delimitado por \0 ou Ctrl+d, que representam EOF (*End Of File*). Cada arquivo pode ser lido caractere a caractere e gerado da mesma forma. Para se referir a um arquivo, o sistema usa os **descritores de arquivos**, que é são palavras chave associadas a cada arquivo. Os sistemas de arquivos serão explicados em maior detalhe numa seção adiante, mas este conceito nos serve para entender um ponto chave do trabalho do *shell*.

O *shell* define três descritores de arquivos muito importantes:

- **a entrada padrão (*stdin*):** como o nome sugere, a entrada padrão é o descritor de arquivo de onde um aplicativo lerá a entrada de dados se não for informado um outro descritor de arquivos específico. Está normalmente associada ao terminal de entrada, normalmente o teclado.
- **a saída padrão (*stdout*):** é o descritor onde será colocada a saída de qualquer aplicativo, se outro específico não for informado. Normalmente associada ao terminal de saída, o vídeo.
- **saída de erro padrão (*stderr):*** é onde são escritos os erros decorrentes do processamento. Está separada do saída padrão para que os possíveis erros ou avisos não contaminem os resultados em si. Também está associada ao terminal de saída.

Neste contexto, as maioria dos comandos do *shell* agem como filtros, que possuem uma única entrada (*stdin*) e duas saídas (*stdout* e *stderr*), por onde entra e sai um caractere por vez. O que cada comando faz é filtrar a entrada, transformando-a e escrevendo-a na saída padrão, enquanto as eventuais mensagens de erro vão para a saída de erros. A Fig. 2 ilustra este conceito.

5 Redirecionamentos

Os recursos de redirecionamento de **entrada e saída** (E/S) são úteis para redefinir as entradas e saídas padrão em outros descritores de arquivos definidos pelo

usuário. A tabela 1 mostra a função dos operadores de redirecionamento para os dois principais tipo de *shells*.

`bash/ksh`	`tcsh`	ação
< ⟨arquivo⟩		considera ⟨arquivo⟩ como *stdin*
> ⟨arquivo⟩		coloca a *stdout* no novo ⟨arquivo⟩
>> ⟨arquivo⟩		anexa *stdout* ao ⟨arquivo⟩ (ou cria-o)
<< ⟨delim⟩		toma como *stdin* até encontrar ⟨delim⟩
2 > &1	> &	reúne *stdout* com *stderr*
2 >> &1	>> &	anexa *stderr* a *stdout*
	>! ⟨arquivo⟩	sobrescreve *stdout* em ⟨arquivo⟩
2 > &1|	|&	reúne *stderr* e *stdout* no duto

Tabela 1: Os operadores de redireção de E/S.

Por exemplo, para criar uma arquivo com a lista dos arquivos de um diretório, faríamos:

`$ ls -l > lista_de_arquivos`

se `lista_de_arquivos` já existisse, poderíamos sobreescrevê-lo[3]

`$ ls -l >! lista_de_arquivos`

ou acrescentar a nova listagem ao final do arquivo

`$ ls -l >> lista_de_arquivos.`

Em todos estes exemplos, ao invés de colocar a saída padrão na tela, todo o seu conteúdo vai para outros arquivos, isto é, estamos redefinindo a saída padrão. A operação análoga pode ser feita com a entrada padrão. Por exemplo, na situação em que quiséssemos mandar um arquivo como conteúdo de uma mensagem de correio eletrônico, simplesmente faríamos:

`$ mail` ⟨endereço⟩ < ⟨mensagem⟩

Por fim, podemos reunir ambos conceitos numa mesma situação em que determinado programa lê seus parâmetros iniciais de um arquivo e grava-os em outro. Para ordenar os nomes dentro de um dado arquivo e gravá-los em outro, simplesmente diríamos:

`$ sort < fora_de_ordem > ordenados`

Neste caso, como em muitos dos aplicativos UNIX, é suposto que o primeiro argumento trata-se da entrada padrão, de forma que poderíamos omitir o sinal de <. Outra aplicação dos redirecionadores é criar um arquivo usando o comando `cat`[4] simplesmente escrevendo

[3]Lembre-se que estamos usando a sintaxe do `tcsh`. Consulte a Tab. 1 se o seu *shell* preferido é outro.

[4]O `cat` foi concebido para concatenar arquivos, isto é
`$ cat` ⟨arq_1⟩ ⟨arq_2⟩ ⟨arq_3⟩ > ⟨arq_final⟩
faz com que ⟨arq_final⟩ contenha exatamente, exceto os marcadores de fim de arquivo, todo o conteúdo de ⟨arq_1⟩ e ⟨arq_2⟩ e ⟨arq_3⟩, nesta ordem. Para ler o manual de qualquer comando, digite `man` ⟨comando⟩.

```
$ cat > astros
mercurio
venus
terra
marte
jupiter
Ctrl+d
```

Aqui o `cat` imprimirá a entrada padrão, que como não foi especificada permanece sendo o teclado, na saída padrão, que agora é o arquivo `astros`. Então podemos entrar a lista de planetas até que Ctrl+d (caractere de fim de arquivo - EOF) seja pressionado. Neste momento, o `cat` grava o nome dos planetas no arquivo.

O recurso dos delimitadores também é util quando o argumento de entrada tem mais de uma linha. Neste exemplo usamos os comando `cat`, que serve para imprimir a entrada padrão na saída padrão sem qualquer formatação, para imprimir as seguintes linhas

```
$ cat << ATE_AQUI
primeira linha
segunda
terceira
ATE_AQUI
```

imprimirá

```
primeira linha
segunda
terceira
```

na saída padrão.

6 Dutos

Os dutos (*pipes*) são elos de ligação entre a saída padrão de um programa e a entrada padrão de outro, conforme mostrado na figura 3. Este conceito simples está por trás desta ferramenta útil e eficiente. Suponha que você quisesse juntar (`cat`) o conteúdo de dois arquivos (`alunos` e `alunas`), ordená-los (`sort`) e gravar a saída num novo arquivo (`turma`). Para isto bastaria

`$ cat alunos alunas | sort > turma`

Desta forma o `cat` concatena `alunos` e `alunas` numa mesma saída padrão, que servirá como entrada padrão para o `sort`, que terá sua saída redirecionada para o arquivo `turma`. O comando anterior sem o uso de dutos seria escrito

`$ cat alunos alunas > alunosealunas`

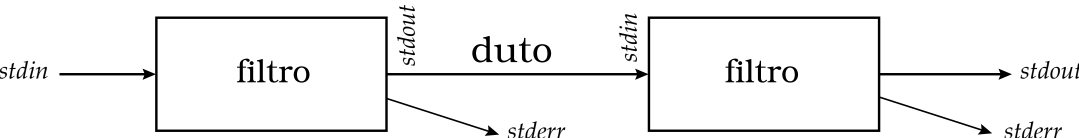

Figura 3: Um duto entre dois filtros.

$ sort alunosealunas > turma

$ rm alunosealunas

Na forma tradicional a quantidade de comandos é maior e há a necessidade de usar o arquivo temporário `alunosealunas`. Também é possível usar mais de um duto na mesma linha de comando. Para imprimir diretamente a saída do processo acima simplesmente acrescente o comando de impressão `lpr`

$ cat alunos alunas | sort | lpr

7 Caracteres Especias e Curingas

Para facilitar o manuseio de arquivos o *shell* dá sentido especial a certos tipos de caracteres que mudam de valor conforme o uso a que se dá a eles, por isso chamados de curingas (formalmente: metacaracteres). Estas regras tendem a reduzir a digitação e encorajar convenções na nomeação de arquivos. Os curingas são os seguintes:

- *****: poderá ser substituído por zero ou mais caracteres.
- **?**: poderá ser substituído por um caractere.
- **[...]**: poderá ser substituído por um dos caracteres entre colchetes.

Por exemplo, se tivéssemos os arquivos `dia1`, `dia2` e `dia3`, poderíamos concatená-los todos usando os curingas apropriados, sem mencionar explicitamente cada um dos nomes. Neste caso, existem três possibilidades de realizar o mesmo trabalho

$ cat dia* > dias

$ cat dia? > dias

$ cat dia[123] > dias

Estes conceitos podem ser generalizados e superpostos, isto é, `?ida*` poderia ser coincidir com `cidade`, mas não `opacidade` nem `claridade`. Da mesma forma, `cor[ear][sle]` passaria por `cores`, `coral` e `corre` (analise as outras combinações possíveis). Além disso, você poderá especificar um intervalo de caracteres, ou seja, a expressão `[a-z]` significa qualquer caractere minúsculo e `[0-9]` qualquer número. Para complementar estas possibilidades, existe o caracteres de negação ^, usado em associação com os colchetes `[...]`. Se escrevemos

`aluno[^0-9]` estamos nos referindo aos nome que começam com a palavra `aluno` **não** seguida dos números de 0 a 9.

Exitem outros caracteres especiais para o *shell* além dos curingas. Um deles é o ponto-e-vírgula `;` que serve para colocar multiplos comandos numa mesma linha. Assim escrevendo

```
$ cd fazenda; rm cavalos* ; cd ..
```

exterminaríamos todos os eqüinos de uma vez e voltaríamos ao diretório anterior.

8 Controle de Processos

Outra característica de operação bastante útil está relacionada ao caractere &. Lembre-se que o Linux, sendo um *sabor* de UNIX, é multiusuário e multitarefa. Isso significa que vários usuários podem estar usando o mesmo computador e executando mais de uma tarefa ao mesmo tempo, com os recursos compartilhados. Assim, se desejássemos colocar um determinado programa para ser executado em segundo plano, anexaríamos o caractere & depois do comando. Imagine que um programa seu chamado `chuva` levará dias executando e você não quer esperar que ele termine ou precisa continuar trabalhando. Bastaria executá-lo como

```
$ chuva &
[1] 925
```

e o *shell* o colocará em segundo plano, informando a ordem dentre os seus processos (`[1]`) e a ordem geral do processo (`925`). O processamento em segundo plano é um recurso poderoso quando bem utilizado, permitindo que preencha-se o tempo ocioso dos computadores, entretanto lembre-se que se o programa não terminar por si próprio ficará rodando até que seja explicitamente morto, como no caso de uma reinicialização ou através do comando `kill`[5]. Uma maneira alternativa de colocar um programa já iniciado em segundo plano é através da sequência de teclas Ctrl+z, ou

```
$ chuva
Ctrl+z
Suspended
$ bg
[1] chuva &
```

pois quando Ctrl+z é pressionado o programa é suspenso até que a ordem `bg` (de *background* ou segundo plano) faz com que sua execução continue em segundo plano, exatamente de onde tinha parado quando o *shell* suspendeu-o, sem nenhuma alteração de seus dados ou parâmetros.

Outra utilidade deste recurso consiste em suspender um aplicativo, realizar outra tarefa e depois continuar executando-o. No exemplo anterior

```
$ chuva
```

[5]Veja o manual do `kill` (`$ man kill`) para maiores detalhes de como matar processos, mas não torne-se um assassino.

```
Ctrl+z
Suspended
$ sol
sol:  Command not found.
$ fg
chuva
```

aqui a ordem `fg` serve para recolocar o ultimo aplicativo que foi para segundo plano (`bg`) para primeiro plano (`fg` de *foreground*). Se há vários processos em segundo plano, é possível indentificá-los antecedendo o número do processo por %, isto é `fg %1`, neste caso.

9 Sustituição da Saída Padrão

A substituição de *stdin* é outro poderoso recurso do *shell* usado através dos acentos graves ` `. Os graves servem para que você coloque a saída da execução de um programa como argumento de outro, de uma maneira diferente daquela com dutos. Por exemplo, o comando `which` (qual) serve para mostrar aonde o *shell* está achando determinado aplicativo. Se fizemos

```
$ which cat
/bin/cat
```

somos informados que o *shell* executa o `cat` que está em `/bin`. Se quisermos saber as propriedades do arquivo `cat`, usamos o `ls`[6] em conjunto com os acentos graves

```
$ ls `which cat`
```

e a saída do `which cat` servirá de argumento para a execução do `ls`. Portanto, a linha anterior seria idêntica a escrever

```
$ ls /bin/cat
```

Novamente realizamos duas operações ao mesmo tempo: achamos o `cat` no sistema de arquivos e listamos suas propriedades. Este exemplo sintetiza a filosofia do UNIX: uma série de programas simples que agrupados proporcionam uma grande eficácia e flexibilidade. A importância do *shell* neste contexto e proporcionar comunicação dos aplicativos entre si e de você com os aplicativos.

10 Variáveis do *shell*

O *shell* permite que se criem variáveis e que se lhes atribuam valores guardados durante a execução do *shell*. As variáveis podem ser **variáveis locais** ou **variáveis de ambiente** (globais). Os nomes das variáveis podem ser constituídos de quaisquer caracteres alfanuméricos. Para criar e ao mesmo tempo atribuir um valor à variável na mesma operação, usamos o comando `set` para

[6] O `ls` lista arquivos e suas propriedades. Voltaremos a falar dele mais tarde.

variáveis locais ou `setenv` para as de ambiente.[7]. No caso do `setenv` não é usado o sinal de = entre o nome e o valor da variável. O valor de uma variável é acessado através do nome da variável precedido pelo operador $[8], por exemplo

```
$ set estado= ¨Rio Grande do Sul¨
$ echo $estado
Rio Grande do Sul
```

Neste caso o `echo` mostra o conteúdo de `$estado` na tela. Note que o argumento passado para o `echo` é "Rio Grande do Sul"pois o *shell* já fez a interpolação da variável `$estado` na cadeia de caracteres correspondente. As variáveis podem ser usadas em qualquer comando no contexto do *shell*. Para referir-se ao nome de uma variável sem ambiguidade, inclue-se seu nome entre chaves { } , observe

```
$ set arquivos=¨/dados/curso¨
$ cp ${arquivos}/aula2.tex /home/alunos
```

assim o *shell* procura por uma variável `arquivo` e concatena o seu valor com o resto da cadeia `/aula2.tex` e então passa o argumento inteiro para o comando `cp`. Note que tanto a interpolação de variáveis em seus respectivos valores como a substituição dos curingas numa lista de arquivos (Seção 7) é feita pelo *shell* e o resultado desta operação (em caso de sucesso) é passada ao aplicativo. Nenhum dos aplicativos realmente recebe `$estado` ou `aluno?` como argumento, mas o significado que estas expressões tem para o *shell*. Se for necessário usar caracteres especiais do tipo `$, *, ?, [ ]` como argumento de aplicativos, usa-se o **caracter de fuga** \ precedendo os caracteres especiais. Isto evita que o *shell* interprete-os. Por exemplo, para lermos a página de ajuda do comando `less`, que serve para mostrar o conteúdo de um arquivo no terminal, usamos `$ less -\?` pois não queremos que o *shell* substitua `?` pelo nome de um arquivo com uma só letra (se `?` fosse um curinga) mas que o `less` receba o `?` como argumento.

O recurso de substituição da saída padrão (Seção 9) pode ser explorado para definir variáveis a partir da saída de aplicativos

```
$ set sistema=‘uname‘
$ echo $sistema
Linux
```

que pode ser extensivamente explorado na construção de scripts do *shell*.

Algumas variáveis são definidas internamente pelo *shell* e podem ser usadas durante sua execução. A tabela 2 relaciona parte das variáveis definidas pelo `tcsh` ao iniciar.

A interpolação de variáveis ocorre sempre que seu nome for invocado diretamente ou entre aspas duplas. Qualquer texto entre aspas simples será interpre-

[7]No caso do `bash` é usado ⟨nome_da_variável⟩=⟨valor⟩, sem a necessidade do `set.` No caso de variáveis de ambiente, depois de definí-la, executa-se o `export` ⟨nome_da_variável⟩ para torná-la global.

[8]Até aqui indicamos o *prompt* do *shell* pelo mesmo símbolo. Não confuda o `$` no início da linha, indicando que se trata da linha de comando, com `$` antes de um nome de uma variável (`$nome`), que serve para que seu valor seja interpolado.

Nome da Variável	Significado
`user` e `USER`	nome do usuário
`home` e `HOME`	diretório padrão de `$user`
`path`	lista de diretórios onde o *shell* procura pelos aplicativos
`shell`	*shell* em uso
`tcsh`	a versão do `tcsh`
`cwd` ou `PWD`	diretório de trabalho atual
`HOST`	nome do computador
`HOSTTYPE`	arquitetura do computador

Tabela 2: Variáveis definidas pelo *shell* `tcsh`. Os nomes em minúsculas são locais e em MAIÚSCULAS são de ambiente.

tado literalmente, sem interpolações. Acompanhe o exemplo abaixo

```
$ set dia=sexta
$ set aviso=¨Ontem foi $dia¨
$ echo $aviso
Ontem foi sexta
$ set aviso='Ontem foi $dia'
$ echo $aviso
Ontem foi $dia
```

desta forma, as aspas simples fornecem um mecanismo para que as interpolações de variáveis sejam ignoradas, enquanto as duplas permitem que as substituições sejam feitas, preservando a unidade da cadeia de caracteres.

11 História de Comandos

A partir do momento em que o *shell* inicia, ele passa a guardar todos os comandos digitados pelo usuário, na sequência em que são digitados. Estes comandos podem ser recuperados, editados e novamente executados pelo *shell*, simplificando tarefas repetitivas em arquivos distintos. Este recurso é chamado de **história** porque grava a sequência de acontecimentos no âmbito do *shell*.

O número de comandos gravados pelo *shell* é determinado pela variável de ambiente `history`[9], enquanto que a variável `savehist` determina quantos comandos serão gravados no arquivo `.history` quando o usuário termina a sessão, para serem recuperados numa sessão futura.

A primeira maneira de examinar os comandos do *shell* é através do comando `history` que mostra em parte algo como

⋮

[9]Por exemplo, set history=100 grava os 100 últimos comandos

```
122   9:35   df
123   9:35   new
124   9:35   rusers
125   9:35   ls
126   9:35   sort Makefile
127   9:35   sort Makefile > lixo
128   9:35   sort Makefile > lixo2
129   9:35   ls
130   9:35   diff lixo lixo2
131   9:35   diff lixo Makefile
132   9:35   ls
133   9:35   clear
134   9:35   rm lixo*
135   9:35   make
            ⋮
```

onde vemos o número do comando na história, a hora da chamada e a sintaxe do comando. O **caracter de substituição da história** é o !, logo se quiséssemos executar o comando de ordem 133 (`clear`) novamente, simplesmente digitamos

```
$ !133

clear
```

que limparia a tela do terminal.

Operador	Efeito
!!	comando anterior
!n	n-ésimo comando
!-n	!n a partir do fim
!`txt`	comando mais recente que inicia com `txt`
!`?txt?`	comando mais recente que contém `txt`
!`*`	todos argumentos do comando anterior, exceto o argumento zero (o nome do comando)
!^	o primeiro argumento do comando anterior
!$	o último argumento do comando anterior
!{`txt`}	faz com que somente `txt` seja usado na interpretação da história

Tabela 3: Indicadores da história do `tcsh`.

Na maioria das implementações do `tcsh` é possível usar as setas do teclado ↑ ↓ para percorrer a história de comandos. Existem alguns modificadores da história que simplificam o reutilização dos comandos já digitados. Para repetir o último comando executado digitamos !!. Como no exemplo acima, o comando de ordem n é reexecutado através da sintaxe !n, onde n é seu número. Se quisermos

contar do fim para o começo, isto é, para reexecutar o comando n a partir do atual, basta fazer $!-n$. Se o `clear` devesse ser executado novamente, poderíamos procurá-lo pelo seu nome, através da sintaxe `!nome`. Neste caso

```
$ !clear
clear
```

fazemos com que o *shell* procure pelo comando mais recente que comece com o texto *clear*, podendo abreviá-lo por `!clea`, `!cle` ou mesmo `!c`. O *shell* lê o os comandos na ordem inversa daquela em que foram digitados, até encontrar o texto que coincida com aquele fornecido pelo usuário, quando executa o comando. Veja outros operadores de procura da história na tabela 3.

11.1 Modificadores e Indicadores

Embora todos estes recursos sejam valiosos para abreviar a quantidade de texto digitado pelo usuário, são pouco frequentes as situações em que uma sequência de comandos é executado várias vezes da mesma maneira[10]. Normalmente deseja-se fazer pequenas modificações nestes comandos e então reexecutá-los. Este é o caso de repetir-se o mesmo processo em vários arquivos, corrigir erros de digitação ou testar novos parâmetros dos aplicativos. Para esse fim o manipulador da história do *shell* inclui os **modificadores da história**. Os modificadores servem para que se repita os comandos da história alterando-o antes de sua execução. Os principais estão listados na tabela 4

Operador	Efeito
`:*`	todos os argumentos, exceto o zero
`:^`	o primeiro argumento
`:$`	o último argumento
$:n$	o n-ésimo argumento
$:p-q$	do argumento p ao q
$:-q$	abrevia $:0-q$
$:p*$	argumento p até o último
`:p`	imprime o comando mas não executa
`:s/l/r/`	substitui `l` por `r`
`:&`	repete a substituição anterior

Tabela 4: Modificadores e indicadores de argumentos da história do `tcsh`.

Os argumentos são nomeados a partir de zero (o nome do comando), conforme o esquema abaixo.

```
$ ls discos livros telas ...
   0    1      2     3   ...
```

[10]Para este fim tem-se o comando interno do *shell* `repeat`

Tomando este comando como exemplo, poderíamos usar os indicadores de argumentos da tabela 4 para selecionar cada um dos argumentos das linha de comando anterior. Por exemplo, !`*` seria substituído por `discos livros telas`. Então, usando !`*` reaproveitamos todos argumentos do `ls` numa nova operação.

```
$ rm !*
rm discos livros telas
```

agora, `rm discos livros telas` é o comando mais recente da história, de maneira que !! referir-se-ia a ele e não mais ao `ls ...`. Para selecionar um só argumento da linha de comando anterior, usamos os indicadores de argumentos. !^ seria substituído por `discos`, !`$` por `telas`, !`:2` por `livros`, !`:2-3` por `livros telas` e assim por diante. Novamente, explore a tabela 4 com criatividade.

Os modificadores e indicadores da história podem ser superpostos, desde que não sejam contraditórios. Além disso, podem referir-se a qualquer comando da história e não só ao último. Nos exemplos acima, bastaria incluir o número do comando logo após o ponto de exclamação ! para que o *shell* refira-se a outro comando que não o último. Se examinamos a história listada no início desta seção (página 10) poderíamos editar o arquivo `Makefile` recém criado, simplesmente digitando

```
$ vi !128:1
vi Makefile
```

se quiséssemos apenas fazer a substituição e colocá-la na história sem executar o comando em si, bastaria acrescentar o `:p` na linha substiuição acima

```
$ vi !128:1:p
vi Makefile            [COMANDO NÃO EXECUTADO]
```

e assim sucessivamente poderíamos empilhar modificadores e indicadores para se referir a comandos passados.

Em suma, o recurso de registro, edição e procura da história facilita o processo de edição de comandos, agilizando o processo de digitação e abreviando a tarefa de memorização de nomes de arquivos. Neste ponto fica difícil evitar a analogia com a vida real, onde também se aprende com os fatos passados. Sem atribuir ao *shell* propriedades da natureza humana, é o conhecimento da nossa história que nos permite evoluir.

12 Apelidos de Comandos

Os apelidos de comandos (*alias*) servem para criar ou redefinir comandos a partir dos comandos do sistema, criando assim um *alias*. Criar um alias é bastante simples, basta usar

`$ alias` ⟨nome do apelido⟩ ⟨comandos⟩

de maneira que quando o ⟨nome do apelido⟩ for chamado, na verdade os comandos que o *shell* executará serão os ⟨comandos⟩. Por exemplo, você pode definir

um comando `apagar` a partir do `rm -i`, onde o `-i` serve para que seja solicitada confirmação ao apagar cada arquivo. Veja

```
$ alias apaga 'rm -i'
```

assim cada vez que o usuário digitar `apaga` o que o *shell* estará executando `rm -i`. Pode-se usar o auxílio de colocar mais de um comando numa mesma linha para definir apelidos. Para criar um apelido `lista` que limpa a tela antes de executar o `ls` e depois mostra a data, bastaria escrever

```
$ alias lista 'clear; ls; date'
```

É aconselhável não definir um apelido com o mesmo nome de um comando já existente, pois podem existir outros apelidos que contenham tal comando na sua definição e desta forma sua definição estará pervertida. Em outras palavras, evite que um apelido seja definido a partir de outro. Acompanhe o exemplo

```
$ alias ls 'ls -laFL'
$ alias dir 'ls'
```

desta forma, `dir` executará `ls`, que por sua vez está definido como `ls -laFl` e não simplesmente `ls`, como era a intenção. Faça pequenas modificações no nome do apelido com relação ao nome do comando, como `ll`, `lls`, etc., para evitar tal efeito colateral

Se existe um apelido com mesmo nome que um comando do sistema, você poderá ignorar a definição do apelido usando o caracter de fuga \ antes do nome do comando

```
$ alias ls 'ls -laFL'
$ \ls
```

executará `ls` e não `ls -laFL`. Estas definições estão entre aspas simples para que o *shell* não tente interpretar qualquer um dos parâmetros entre aspas. Se você deseja que o *shell* faça as substituições necessárias, reescreve sua definição usando aspas simples ". Por exemplo o comando

```
alias bomdia ¨cat $HOME/mensagem¨
```

definirá `bomdia` como `cat /home/ferrari/mensagem` no meu caso.

O uso dos apelidos ajuda a recordar comandos extensos e complexos, além de resumir a quantidade de digitação. Em contrapartida, o abuso no uso dos apelidos faz com que o usuário perca ciência do que realmente está fazendo, um risco a longo prazo.