FMA 215

Aula 7: Usando a Shell

Prof. Jorge L. deLyra

06 de Maio de 2002

1 Alguns Conceitos Relevantes

Shell: a shell é um programa que está rodando continuamente em suas seções interativas, estabelecendo a comunicação entre o usuário e o sistema. Já estivemos usando a shell em todas as tarefas anteriores sem, entretanto, dar a ela especial atenção, que é o que faremos aqui. As shells modernas têm uma série de sistemas e dispositivos internos que facilitam e tornam mais ágeis e rápidas as atividades do usuário no sistema. Elas dispõem também de estruturas de programação que podem ser utilizadas diretamente pelo usuário na linha de comando, para automatizar as mais diversas tarefas.

Através destes procedimentos de programação de linha de comando e do uso de algumas propriedades padronizadas que valem para a maior parte dos comandos e programas do sistema, a shell permite que combinemos a ação dos muitos comandos do sistema de formas arbitrariamente complexas, para criar novas funções e capacidades. O ambiente de uso e a filosofia de operação dos sistemas Unix e, em particular, do Linux, leva o usuário, de forma natural, a se habituar a pensar um pouco como um programador, a pensar em como fazer coisas que não estão explicitamente previstas pelas estruturas básicas do sistema. Isto enriquece e torna muito mais fértil e poderoso o uso do sistema para as mais variadas tarefas de todo tipo.

Wildcards: as wildcards ou "curingas" são caracteres que são interpretados de forma especial pela shell. O seu uso permite que você se refira a conjuntos de arquivos de uma só vez, em comandos que estiver executando. Existem duas ou três wildcards de uso muito difundido para se lidar com arquivos, em especial os caracteres "*" e "?". Eles podem ser usados em combinação com qualquer comando que se use para manipular arquivos.

O conceito de wildcard é extendido e enriquecido, no âmbito de alguns comandos, para o conceito de "regular expressions", que é mais extenso e poderoso, podendo ser utilizado para se lidar com texto de formas mais complexas. Os comandos que usam estas regular expressions incluem os editores `vi` e `emacs`, um editor muito especial chamado `sed`, o comando de procura textual `grep`, os paginadores `more`, `less` e `most` e a linguagem de programação `awk`. Veremos alguns destes programas mais tarde.

Variáveis: Há dois tipos de variáveis no sistema, as variáveis de ambiente (environment) e as variáveis de shell. As variáveis de environment são mantidas pelo próprio sistema e estão disponíveis para todas as shells, de qualquer tipo. Estas variáveis têm nomes inteiramente em letras maiúsculas. Quando uma shell de login é iniciada ela herda do sistema um determinado conjunto de variáveis básicas de environment. Quando uma shell é iniciada a partir de outra shell, ela herda daquela shell todas as variáveis de environment que estiverem definidas. A partir da shell é possível criar, modificar e eliminar variáveis de environment.

As variáveis de shell são definidas e mantidas por cada shell, estando disponíveis apenas para o processo daquela shell em particular. Em geral estas variáveis têm nomes inteiramente em letras minúsculas. O processo de inicialização da shell que é realizado cada vez que uma nova shell é iniciada, seja ela uma shell de login ou não, em geral define algumas variáveis básicas que controlam o comportamento da shell. De dentro da shell pode-se definir, modificar ou eliminar quaisquer variáveis de shell. Um dos usos mais importantes das variáveis de shell é em programas e procedimentos de linha de comando, bem como em programas implementados através de shell scripts.

Shell Scripts: são programas que podemos escrever em arquivos do tipo texto, numa linguagem própria da shell. Estes arquivos são lidos pela shell e os comandos são interpretados e executados por ela. Trata-se de um sistema de programação interpretada, como é típico do BASIC, não compilada, como é típico do Fortran e do C. Se por um lado isto faz com que este tipo de programação seja relativamente lento do ponto de vista da execução, por outro lado isto não é muito importante para as funções a que se destina a maior parte dos programas que são escritos desta forma.

Este tipo de programação tem enormes vantagens para as atividades de programação e gerenciamento do sistema, bem como para muitas funções de interesse para os usuários, envolvendo a manipulação de texto. Ele é extremamente flexível, altamente portável e pode-se escrever os programas com muita rapidez. Não só todas as principais shells que existem (a `tcsh` e `bash` são as mais populares) têm linguagens de programação deste tipo, como hoje em dia há várias shells especializadas, que são usadas apenas para escrever programas deste tipo como, por exemplo, a `perl` e a `python`.

Na realidade, estes programas, assim como o `awk`, são melhor descritos como interpretadores de certas linguagens de programação do que como shells, pois não podem ser usadas interativamente para o acesso ao sistema, que é uma das funções básicas das shells.

2 Tarefas

A shell padrão que utilizaremos é a `tcsh` (Tenex C Shell), uma versão muito melhorada e extendida da tradicional shell `csh` (C Shell) do Unix. É esta a shell que usamos como padrão em todos os nossos sistemas. Algumas das características e estruturas que vamos examinar são específicas desta shell, mas a maior parte tem análogos muito próximos e com funcionamento muito parecido nas outras shells mais populares como, por exemplo, a `bash` (Born Again Shell) uma versão muito melhorada e extendida da tradicional shell `sh` (Bourne Shell) do Unix.

A shell tem um grande número de estruturas e sistemas muito úteis que estão à sua disposição para facilitar a sua interação com o sistema. Nesta aula nós vamos proceder a uma exploração geral de algumas das principais estruturas e sistemas, sem pretender nos aprofundar no uso mais sofisticado de todas as possibilidades existentes. Vamos nos limitar aqui ao uso interativo básico da shell, sem considerar procedimentos de programação na linha de comando. Mais adiante haverá outras aulas, especificamente para tratar de programação de linha de comando, bem como de shell scripts.

Para seu maior conforto, é melhor fazer estas tarefas no sistema de janelas X11, mas também é possível realizá-las em um terminal de caracteres.

1. Uma das coisas úteis que a shell pode fazer é completar para você, de forma automática, os nomes de comandos e arquivos que você começa a digitar. Para solicitar uma operação de completamento, basta digitar

a tecla [Tab]. Se você já tiver digitado parte do nome de um comando, uma das seguintes coisas vai acontecer:

   (a) se houver um único comando no sistema com o início do nome igual à parte que você já digitou, o nome será completado;
   (b) se houver mais de um comando com este mesmo início de nome, será mostrada uma lista das possíveis alternativas;
   (c) se não houver nenhum comando com este início de nome, você ouvirá um “beep” no altofalante do computador.

   Para testar isto, aumente ao máximo o seu terminal `xterm`, digite o caracter `a` seguido de [Tab] e veja o que acontece. Tente o mesmo com várias outras letras ou combinações de letras.

2. O comportamento da shell também é parecido com este no caso do completamento de nomes de arquivos. Se você já tiver digitado todo o nome do comando e estiver digitando o nome de um arquivo como argumento do comando, um [Tab] fará com que a shell tente completar o nome do arquivo para você. Se você iniciar o nome sem um caracter `/`, ou seja, usar um nome de arquivo relativo, o arquivo será procurado pela shell no seu diretório corrente. As alternativas relativas ao que pode acontecer quando você faz isto são as mesmas de antes. Isto é conveniente para lidar com arquivos cujos nomes sejam muito grandes. Crie um arquivo vazio com um nome ridiculamente grande como

   `este-e-um-arquivo-vazio-com-um-nome-ridiculamente-grande`

   Em seguida execute o comando ls -l <arquivo> usando este arquivo como argumento. Após ter digitado o `ls`, provavelmente basta digitar `este-` e em seguida digitar [Tab].

3. No caso do completamento de nomes de arquivos também aparece uma lista de alternativas possíveis, o que também é muito útil. Imagine, por exemplo, que você queira ir para algum subdiretório da raiz do sistema mas tenha esquecido o seu nome. Digite `ls /` e tecle em seguida [Tab]. Quando aparece a lista, muitas vezes basta digitar uma ou duas letras iniciais do nome do diretório apropriado e mais um [Tab] para ser levado para o lugar certo, mas nem sempre, às vezes é necessário digitar a maior parte do nome. Tente fazer isto, indo para o diretório `/etc/init.d/`. Em seguida, volte para casa.

4. Uma outra coisa muito útil que a shell faz é lembrar os comandos que você usou anteriormente. Esta estrutura é chamada de `history`. Existe também um comando com este nome, que mostra todos os comandos já executados. Use-o para ver todos o comandos que você usou até agora. É provável que você tenha de usar o comando combinado com um paginador, como por exemplo `history | more`, para poder ver todos os comandos, pois a lista completa provavelmente já está um tanto longa.

5. É possível passar pela lista de `history` de comando em comando usando as teclas de cursor, aquelas marcadas apenas com setas apontando para cima, para baixo, para a esquerda e para a direita. Vamos denotar estas teclas por `[↑]`, `[↓]`, `[←]` e `[→]`, respectivamente. Se for o caso de repetir um comando, basta usar a tecla `[↑]` para passar sucessivamente para cada comando anterior, enquanto a tecla `[↓]` volta para os comandos mais recentes. Uma vez encontrado o comando que se quer, é possível executá-lo novamente com a tecla `[Enter]`. Faça isto com alguns dos comandos que você já usou antes.

6. Também é possível modificar os comandos da `history` antes de executá-los novamente, editando-os de forma simples com o uso das teclas `[←]`, `[→]` e `[BackSpace]`. As teclas de seta para a esquerda e para a direita farão com que o cursor do teclado caminhe nas respectivas direções sobre o texto do comando que foi lembrado, enquanto a tecla `[BackSpace]` pode ser usada para apagar caracteres para trás. Use o sistema de `history` para voltar ao comando que usou anteriormente para criar um arquivo com o nome ridiculamente longo e edite o comando antes de executá-lo novamente, para criar um novo arquivo com o nome ridiculamente longo, tal como

   ```
   este-e-outro-arquivo-vazio-com-um-nome-ridiculamente-longo
   ```

7. Observe que, enquanto você está editando um comando, você pode passar do modo "insert" para o modo "overwrite" e de volta, digitando repetidamente a tecla `[Insert]`. Isto muda a forma como os novos caracteres que você digita são inseridos na linha de comando. No modo insert eles são colocados onde estiver o cursor e os caracteres à direita dele são movidos para a direita para abrir espaço. No modo overwrite, os novos caracteres são escritos sobre aqueles que estão à direita do

cursor. Dois outros comandos úteis na edição da linha de comando são as combinações de teclas `[Ctrl]-A`, que leva o cursor para o início da linha, e `[Ctrl]-E`, que leva o cursor para o fim da linha. Estes dois comandos seguem a sintaxe do editor de texto `emacs`, como veremos mais adiante. Para verificar como isto funciona, edite novamente o comando anterior e experimente as várias possibilidades. Caso você não queira de fato voltar a executar o comando, a combinação de teclas `[Ctrl]-C` cancela o comando no qual você está trabalhando.

8. Quando se usa wilcards, em geral a shell faz as substituições com base nos arquivos que existem no diretório onde você está. A wildcard "`*`" representa qualquer conjunto de caracteres seguidos, com a exceção do caracter "espaço em branco" mas incluindo o conjunto vazio. Para experimentar com esta wildcard, vá para o seu home e digite `echo *`. Compare o resultado com o output do comando `ls`. Você compreende qual foi a substituição que a shell fez, para que o resultado seja aquele que de fato aparece?

9. Para tornar as coisas mais claras, tente digitar também `echo .c*` e compare com o output de `ls -a`. Tente usar também `ls *` e `ls .c*` e compare os resultados. Você entende completamente a diferença entre o que os comandos `echo` e `ls` fazem quando usados desta forma? Verifique a diferença entre os outputs de `echo .` e `ls .`, que é o mesmo que simplesmente `ls`. Havendo necessidade, faça pipelines do output destes comandos pelo paginador `more`.

10. A wildcard "`?`" é usada para representar um único caracter, que pode ser qualquer um exceto o espaço em branco, mas não pode ser o conjunto vazio. Para experimentar esta wildcard, tente as combinações `echo .?`, `echo .??`, `echo .???`, `echo .????`, `echo .?????`, `echo .??????` e `echo .???????`. Tente também `ls -l .?` e `ls -l .?????` para ver o que acontece. Finalmente, tente a combinação `ls -l .??*` das duas wildcards.

11. Existe mais um tipo de wildcard que é muito útil, pois ele permite que nós discriminemos explicitamente um ou mais caracteres de uma string de caracteres, deixando os demais arbitrários. Isto é feito através do uso dos brackets quadrados `[ ]`. A combinação `[abc]` pode ser substituída por um único caracter que seja um dos três que estão discriminados

entre os brackets. Para experimentar com esta estrutura, tente as combinações `ls -l .c*`, `ls -l .l*` e `ls .[cl]*`.

12. Também é possível discriminar intervalos de caracteres, ordenados segundo a sequência ASCII que, para letras, coincide com a ordem alfabética. Isto é feito colocando-se um hífen entre dois caracteres dentro dos brackets quadrados. Para experimentar com esta alternativa, tente as combinações `echo .[A-Z]*`, `echo .[a-z]*`, `echo .[A-z]*`, `echo .[a-Z]*` e `ls -l .[c-l]*`. Além disso, é possível colocar entre os brackets vários intervalos de caracteres, separados por vírgulas. Tente, por exemplo, `ls -l .[A-Z,a-z]*` e `ls -l .[X,c-l]*`.

13. Vá para a raiz da conta de algum outro usuário, por exemplo para `/home/delyra` e, uma vez lá, tente executar de novo todos os comandos envolvendo wildcards que você acaba de executar em sua própria conta. Em geral a maior parte das contas de outros usuários estará aberta para leitura, apesar de que você não poderá escrever nelas. Não há nada de errado com isto, de fato esta é uma das formas de se promover a livre troca de informações entre os usuários do sistema. Se quiser, cada usuário pode fechar, protegendo contra leitura por outros usuários, quaisquer partes de sua conta.

14. Outra estrutura de interesse da shell é a possibilidade de definir aliases, ou seja, outros nomes para comandos que já existam no sistema. Assim você pode associar a determinados comandos nomes mais curtos ou que sejam mais fáceis de lembrar por qualquer motivo. Para ver um exemplo disto, execute `alias dir`. Dê uma olhada na página de manual do comando `ls` para ver o que a opção `-C` significa. O comando `alias <nome>` mostra a que comando corresponde de fato o nome discriminado, caso ele seja um alias. Naturalmente, muitas pessoas estão habituadas a usar o comando `dir` para listar arquivos, pois ele é popular em outros sistemas. Com este alias, torna-se a vida destas pessoas mais fácil no Linux, onde o comando que tem esta função é o `ls`.

15. Outra possível utilidade deste tipo de estrutura é dar ao usuário inexperiente (ou a todos os usuários) uma proteção adicional, por exemplo contra a perda acidental de arquivos por se copiar ou movimentar um arquivo em cima de outro que já exista. Para ver algumas das proteções que estão instaladas no sistema contra a perda acidental de arquivos,

execute os comandos `alias cp`, `alias mv` e `alias rm`. Olhe as páginas de manual destes comandos para identificar a função da opção `-i` que aparece nestes aliases. Observe que, neste caso, o nome dos aliases são os mesmos dos comandos originais. A shell sempre procura e interpreta aliases antes de executar arquivos executáveis contendo comandos. Para usar o comando original quando há um alias com o mesmo nome, basta usar o nome do comando precedido de um `\`, como por exemplo em `\rm`.

Mas cuidado, o exemplo que acabamos de mostrar pode ser muito perigoso! O mesmo cuidado deve ser tomado para os outros casos mencionados. O comando `\rm *` apagará instantaneamente todos os arquivos do diretório onde você estiver, sem hesitação e sem fazer perguntas. **Só faça este tipo de coisa se estiver absolutamente certo do que está fazendo!**

16. Já existe uma grande quantidade de aliases definidos no sistema, como parte da configuração de ambiente de usuários que implementamos nele durante a instalação. O comando `alias` usado sem argumentos mostra uma lista completa de todos os aliases que existem. Dê uma olhada no conjunto completo usando a pipeline de comandos `alias | more`. Você verá que há aliases definindo várias versões dos comandos mais usados com os nomes mais populares de outros sistemas. Verifique se você consegue entender a função de cada um dos aliases existentes.

17. Você pode criar os seus próprios aliases usando o comando `alias`. Usado com dois argumentos, da forma `alias <nome> <comando>`, o comando `alias` atribui o novo nome ao comando especificado. O comando pode incluir opções e argumentos, pode até mesmo ser uma pipeline de comandos. Defina um alias que implemente o comando "xtop" que foi discutido anteriormente, executando o comando

    ```
    alias xtop xterm -e top
    ```

    Em seguida execute `alias xtop` para verificar que a operação foi bem sucedida. Por fim, experimente usar o novo comando executando `xtop` para ver se ele de fato funciona. Lembre-se que para sair do `top` basta digitar `q`.

18. Já vimos anteriormente que o comando `env` mostra todas as variáveis de environment e seus valores. Podemos também obter e usar os valores de variáveis individuais, usando o nome da variável precedido do caracter `$`. Por exemplo, o valor da variável `USER` pode ser obtido usando-se `$USER`. Para verificar isto, execute o comando `echo USER = $USER`. Faça o mesmo com as variáveis `HOME`, `TERM`, `SHELL` e `DISPLAY`. Estas são algumas das variáveis mais importantes do environment básico do sistema.

19. Na `tcsh` podemos manipular as variáveis de environment usando o comando `setenv`. Primeiramente, execute `echo $TEST` para verificar que ainda não há uma variável com este nome. Em seguida execute o comando `setenv TEST`, seguido mais uma vez de `echo $TEST`. Observe o que acontece. O comando `setenv` usado com um único argumento cria a variável mas não atribui a ela nenhum valor. Execute agora o comando `setenv TEST 'Este e um teste'`, seguido mais uma vez de `echo $TEST`. Observe a necessidade do uso das aspas, pois o comando `setenv` aceita apenas dois argumentos, na forma `setenv <variável> <valor>`, de forma que é necessário passar o segundo argumento na forma de uma única string, incluindo os espaços em branco. Para terminar, execute o comando `unsetenv TEST`, seguido mais uma vez de `echo $TEST`.

20. Como vimos antes, o comando `set` sem argumentos nos mostra todas as variáveis da shell. Assim como no caso das variáveis de environment, podemos obter e usar os valores de variáveis de shell individuais usando seu nome precedido de um caracter `$`. Para verificar isto, execute `echo user = $user`. Faça o mesmo com as variáveis `home`, `term` e `shell`. Como você pode ver, a shell define e mantém algumas variáveis com funções idênticas às correspondentes variáveis de environment, obtendo dele os valores apropriados para definir estas variáveis. Existem, entretanto, várias outras. Execute o mesmo comando com as variáveis `uid`, `tty`, `group` e `gid`. Para ilustrar os significados de algumas destas variáveis, bem como exercitar o uso de variáveis de um modo geral, execute o comando

    ```
    echo O usuario $user e identificado pelo numero $uid
    ```

    Faça algo parecido com as variáveis `group` e `gid`.

21. Existem também variáveis cujos valores não são fixos, sendo modificados pela shell dependendo das circunstâncias. Execute, por exemplo, o comando `echo cwd = $cwd`. Em seguida, mude para algum outro diretório e repita o comando. Observe que o nome "cwd" é uma abreviação de "current working directory". Uma outra variável importante deste tipo é a variável `status`. Para verificar o que ela faz, execute o comando `true` seguido imediatamente de `echo $status`. Em seguida, execute o comando `false` seguido mais uma vez de `echo $status`. Para entender a diferença, pode ser útil dar uma olhada nas páginas de manual dos comandos `true` e `false`.

22. Na `tcsh` podemos manipular as variáveis de shell usando o comando `set`. Primeiramente, execute `echo $test` para verificar que não há uma variável com este nome. Em seguida execute o comando `set test`, seguido mais uma vez de `echo $test`. O comando `set` usado com um único argumento cria a variável mas não atribui a ela nenhum valor. Execute agora o comando `set test = 'Este e um teste'`, seguido mais uma vez de `echo $test`. Tente executar também o mesmo comando sem as aspas, `set test = Este e um teste`, seguido mais uma vez de `echo $test`. Observe a diferença de comportamento em relação às variáveis de environment. Para terminar, execute o comando `unset test`, seguido de `echo $test`.

23. As variáveis de shell da `tcsh` têm uma propriedade adicional, muito útil, que as variáveis de environment não têm: elas podem ser definidas como "arrays" ou vetores, com um conjunto de valores em vez de um único valor. Execute a pipeline `set | grep argv` para filtrar da lista completa produzida pelo comando `set` a linha contendo a descrição da variável `argv`. Faça o mesmo com a variável `path`. Observe que há algo de diferente em relação a elas, por comparação com as outras. Trata-se de variáveis do tipo "array". A variável `argv`, em particular, é muito útil, seus valores são os argumentos de linha de comando, os argumentos que se coloca em um comando que é executado na linha de comando. Vamos usar muito esta variável quando estivermos explorando o mundo dos shell scripts.

24. Por enquanto, para compreender como lidar com variáveis do tipo array, vamos criar uma nós mesmos. Para começar, repita o comando

```
set test = 'Este e um teste'
```

seguido desta vez de `echo $test[1]` e de `echo $test[2]`. A variável `test` foi criada com um único elemento ou "word", que é acessado pelo uso de `$test[1]`, no estilo típico da linguagem C. Execute agora o comando

```
set test = ( Este e um teste )
```

seguido desta vez de `echo $test`. Aparentemente não há nenhuma diferença em relação ao caso anterior, onde usamos aspas em vez de parêntesis. Entretanto, tente agora executar `echo $test[1]`, `echo $test[2]`, `echo $test[3]` e `echo $test[4]`. Desta vez a variável `test` foi transformada em uma array com quatro palavras. Enquanto `$test` representa o valor de todas as palavras, `$test[n]` representa o valor da `n`-ésima palavra. Para terminar, execute mais uma vez o comando `unset test`, seguido de `echo $test`.

25. Vimos que podemos usar aspas simples para definir strings de caracteres, como em `'Esta e uma string'`. Essencialmente o mesmo efeito pode ser obtido com aspas duplas, `"Esta e uma string"`. Para ver a diferença que há entre o uso de um tipo ou outro de aspa, execute os comandos `echo '$USER'` e `echo "$USER"`. Tente executar também os comandos `echo "'$USER'"` e `echo '"$USER"'` para ver o que acontece. Finalmente, tente repetir os exemplos acima trocando o `$` por `\$`. Você pode combinar estas estruturas de várias formas diferentes para obter diferentes resultados.

26. É possível usar também aspas reversas, que têm um papel completamente diferente. Se colocarmos um comando ou uma pipeline de comandos dentro de aspas reversas, a shell tentará executar o que está ali dentro, como se tivesse sido digitado diretamente na linha de comando. O resultado da operação resultante é substituído, como uma string, no local onde estão as aspas reversas e o seu conteúdo. Por exemplo, vá para o seu home e execute o comando `\ls`, que produzirá uma listagem de arquivos sem cores. Em seguida, atribua esta lista a uma variável chamada `files` executando o comando

```
set files = `\ls`
```

Para verificar o resultado, execute `echo $files`. Em seguida, execute `echo $files[n]` para valores apropriados de `n`. Deve estar claro que este tipo de coisa será muito útil mais tarde, quando estivermos programando com a linguagem da shell.

3 Problemas e Dicas

1. Construa uma lista completa de todos os comandos existentes no sistema. Dicas: lembre-se de que a shell é um programa como qualquer outro; considere combiná-la com o comando `tee <arquivo>`; olhe a página de manual do comando `tee`.

2. Invente uma combinação de wildcards que selecione única e exclusivamente arquivos de configuração da raiz de sua conta, cujos nomes comecem por um ponto e terminem com `rc` (uma abreviação de "run commands").

3. Execute o comando ``ldd `which grep` `` e procure entender o significado do resultado que aparece. Dica: leia as páginas de manual dos comandos `which` e `ldd`.

4. Digite algum comando errando propositadamente o seu nome pela troca ou omissão de um caracter e verifique o que a shell faz a respeito disto. Verifique quais são as suas opções de resposta à pergunta feita pela shell, tente cada uma delas para ver o que acontece.

4 Uma Lista de Comandos Relevantes

`alias <nome> <pipeline-de-comandos>` : atribui um nome à pipeline de comandos; é um comando built-in da shell.

`dir` : um alias para o comando `ls`.

`echo <string>` : ecoa o string de caracteres na tela; é um comando built-in da shell.

`env` : mostra as variáveis de ambiente da sua sessão; é um comando built-in da shell.

`false` : é um comando que não faz nada e ainda assim consegue falhar nisto; o objetivo é simular o status de saída de um comando que falha.

`grep <pattern> <arquivo>` : procura o pattern no arquivo, retornando aquelas linhas do arquivo que contêm o pattern.

`history` : mostra a lista de comandos executados; é um comando built-in da shell.

`set [<variável>] [= (<valor>...<valor>)]` : sem argumentos, mostra todas as variáveis definidas na sua shell; com argumentos, atribui os valores a cada uma das palavras (words) da variável; se só houver um valor pode-se omitir os parêntesis; é um comando built-in da shell `tcsh`.

`setenv [<variável>] [<valor>]` : sem argumentos, mostra todas as variáveis definidas no seu environment; com argumentos, atribui o valor à variável; é um comando built-in da shell `tcsh`.

`true` : é um comando que não faz nada mas que pelo menos consegue ser bem sucedido nisto; o objetivo é simular o status de saída de um comando que é executado com sucesso.

`unset <variável>` : elimina uma variável da shell; é um comando built-in da shell `tcsh`.

`unsetenv <variável>` : elimina uma variável do environment; é um comando built-in da shell `tcsh`.

5 Arquivos de Configuração

Como já sabemos, o comportamento da shell `tcsh` é controlado pelos arquivos de configuração `.cshrc` e `.login`. Em particular, algumas propriedades da shell que vimos nesta aula são controladas pelo valor de determinadas variáveis, em geral definidas dentro de `.cshrc`. A capacidade de listar automaticamente alternativas de completamento quando se digita `[Tab]` é controlada pela variável `autolist`. Verifique a situação desta variável em sua sessão, usando o comando `set`, que é um comando intrínseco da shell. Mude esta situação usando `unset` e veja que consequências isto tem.

O sistema de correção automática de comandos ou nomes de arquivos errados é controlado pelas variáveis `autocorrect` e `correct`, verifique a situação delas e tente modificar esta situação para ver o que acontece. Outra estrutura que você pode controlar é a "history", que é controlada pelas variáveis `history` e `histdup`. Brinque um pouco com estas também. Outras variáveis que controlam aspectos importantes do comportamento da shell são `ignoreeof`, `noclobber` e `notify`. Veja se você descobre alguma coisa sobre estas. Olhar o manual da tcsh pode ser uma boa idéia.