FMA 215

Aula 12: Programação de Linha de Comando

Prof. Jorge L. deLyra

27 de Junho de 2002

1 Alguns Conceitos Relevantes

Além de ter a sua utilidade básica de rodar programas e estabelecer a interface de comunicação entre o usuário e o sistema, a shell pode ser usada para atividades de programação, o que é particularmente útil para a programação da própria estrutura do sistema, mas que também pode ser muito útil para as atividades dos usuários na linha de comando. Como já vimos antes, podemos definir, manipular e usar variáveis dentro da shell. Além disso, ela contém várias estruturas de controle que nos permitem conceber e realizar operações complexas, como veremos nesta apostila. Estas variáveis e estruturas de controle podem ser combinadas com quaisquer dos comandos do sistema, incluindo comandos e programas que o próprio usuário tenha escrito, para a criação de programas de shell que, através da manipulação de texto ASCII, realizem as mais variadas tarefas dentro do sistema.

Assim, vista como uma linguagem de programação, a estrutura da shell é um tanto diferente da estrutura de uma linguagem de programação tradicional. Apesar de ter um número fixo de estruturas de controle, ela pode ter um número ilimitado de comandos disponíveis, pois pode fazer uso de qualquer comando já existente ou que os usuários venham a incluir no sistema. Estes novos comandos devem obedecer apenas ao simples critério de ler texto do canal padrão de entrada `stdin` e escrever texto para o canal padrão de saída `stdout`. Como eles podem ser escritos em quaisquer das linguagens de programação tradicionais, tais como C, Pascal e Fortran, todas as possibilidades destas linguagens estão, em princípio, à disposição do programador de shell. Entretanto, como veremos, com um pequeno número de comandos muito básicos já existentes no sistema é possível escrever uma infinidade de programas para os mais variados fins.

Por outro lado a linguagem da shell, sendo uma linguagem de programação interpretada e não compilada como é o caso das já citadas linguagens tradicionais de programação, não se presta para a otimização em termos de tempo de processamento. Ela se presta bem para a escrita de programas complexos nos quais a preocupação com eficiência de execução não seja tão grande quanto a preocupação com a praticidade e rapidez para se escrever os programas. Todas estas possibilidades de uso da shell podem ser implementadas através de arquivos que são lidos e executados por ela, os assim-chamados shell scripts, como veremos mais tarde. Neste caso, trata-se de uma verdadeira atividade de programação, na qual estes scripts são os programas. Quase toda a programação do sistema é feita desta forma, como se pode ver nos arquivos que estão no diretório `/etc/init.d/`, contendo em sua maioria scripts de `bash`.

2 Tarefas

Antes de explorar os shell scripts, o que faremos em aulas posteriores, nesta aula vamos simplesmente usar as estruturas de controle da shell diretamente na linha de comando, para efetuar de forma simples e fácil operações que, de outra forma, seriam laboriosas e tediosas. Por simplicidade de formulação, em sua forma atual estas tarefas devem ser realizadas em uma sessão interativa no sistema socrates.

Existem dois aspectos desta atividade, sendo um deles simplesmente conhecer as estruturas de controle, as estruturas de dados e os comandos básicos que estão disponíveis. O outro aspecto é a atividade de programação propriamente dita: a análise dos problemas que se tem de resolver seguida da invenção de algoritmos contendo estruturas e sistemas de operações que possam resolvê-los e, finalmente, a escrita dos correspondentes programas.

Nas tarefas abaixo tentaremos tocar em ambos estes aspectos, mas a habilidade de programação propriamente dita não é algo que possa ser desenvolvida em uma única aula. Trata-se de um treinamento que deve ser desenvolvido devagar e a longo prazo. Todo usuário de um sistema moderno, aberto e poderoso como o Linux deve se habituar a pensar pelo menos um pouco como um programador. Comecemos o quanto antes!

Estaremos examinando aqui a linguagem de programação da shell que usamos como shell default do sistema, a `tcsh`. Tudo que vamos fazer aqui também pode ser feito na `bash` através de operações semelhantes, mas com detalhes de sintaxe um pouco diferentes. Você deve abrir uma sessão no socrates e tentar executar nesta sua sessão todos os exemplos e alternativas que são descritos abaixo.

1. A forma mais simples de programação de linha de comando é pouco mais que a simples utilização dos comandos do sistema. Por exemplo, para obter o nome completo de um usuário do qual se conhece apenas o username, pode-se fazer algo como a pipeline

   ```
   grep $user /etc/passwd | cut -d :  -f 5
   ```

   onde estamos usando o seu próprio username como exemplo, que é obtido como o valor da variável `user`. Trata-se de um processo em duas etapas, primeiro obtemos a linha correspondente ao usuário `$user` na base de dados `/etc/passwd`, com

   ```
   grep $user /etc/passwd
   ```

   e, em seguida, cortamos da linha resultante a informação que queremos, com o comando `cut`, usando o caracter "`:`" como separador de campos e selecionando o quinto campo. Tudo o que estamos fazendo aqui é usar os comandos `grep` e `cut` do sistema, além da pipeline, ou seja, a capacidade da shell de concatenar vários programas através de seus inputs (`stdin`) e outputs (`stdout`).

   Nas tarefas e exemplos desta aula que envolvem pipelines de comandos você deve construir a estrutura completa aos poucos, experimentando cada parte dela na linha de comando e acrescentando uma nova operação de cada vez. Neste primeiro exemplo temos apenas dois passos neste processo,

   ```
   grep $user /etc/passwd
   ```

   e

   ```
   grep $user /etc/passwd | cut -d :  -f 5
   ```

   Sempre que você for criar uma pipeline de comandos para realizar uma determinada tarefa você deve proceder desta forma, trata-se de um tipo de programação cujo desenvolvimento envolve extensivamente operações de tentativa e erro.

2. Não é muito mais difícil construir uma lista contendo todos os nomes de todos os alunos que são usuários do sistema. Como definimos, no Projeto Sócrates, que os números de usuário dos estudantes começam a partir de `30001`, podemos usar este fato para selecionar as entradas do arquivo `/etc/passwd` que correspondem aos estudantes. Basta fazer

   ```
   grep ':3[0-9][0-9][0-9][0-9]:'  /etc/passwd | more
   ```

   O alvo de procura do comando `grep` é, neste caso, um pouco mais complexo, utilizando uma das estruturas da classe de estruturas que denominamos de "expressões regulares" ("regular expressions" ou, abreviadamente, regexps)[1]. A expressão `[0-9]` significa qualquer dígito de `0` a `9`. Assim, esta regexp representa qualquer número de `30000` a `39999` circundado de dois caracteres "`:`", o que serve para selecionar os números `uid` dos alunos. A partir desta lista, basta cortar a informação solicitada, usando o comando `cut` como antes:

   ```
   grep ':3[0-9][0-9][0-9][0-9]:' /etc/passwd \
       | cut -d : -f 5 | more
   ```

   Neste último exemplo, para evitar linhas excessivamente longas, estamos usando o caracter `\` para quebrar a linha em duas, sem que o conjunto das duas linhas deixe de ser interpretado como uma única linha pela shell, do ponto de vista lógico. De fato, o caracter `\` tem, dentro da shell, o significado de um "quote", significando que o próximo caracter não deve ser interpretado pela shell da forma usual. Neste caso é o caracter `^J`, que marca o final da linha nos sistemas da família Unix, que deixa de ser interpretado, tornando-se equivalente a um espaço em branco para a sintaxe da shell, de forma que as duas linhas mostradas passam a valer como uma única linha.

3. Como você pode verificar no exemplo anterior, a primeira linha do output não corresponde a um aluno de verdade, trata-se do uid `30000`, que é da conta chamada `proaluno`, utilizada pelos monitores da Sala Pró-Aluno do IF. Para eliminar esta primeira linha, podemos incluir na pipeline o comando `tail +2`, que filtra a parte final do stream de dados que passa pela pipeline, começando na segunda linha.

[1]Uma das melhores referências para você se informar melhor sobre as regexps é na página de manual chamada `perlfaq6`.

```
grep ':3[0-9][0-9][0-9][0-9]:' /etc/passwd \
    | tail +2 | cut -d : -f 5 | more
```

Observe como, ao construir aos poucos uma pipeline, podemos acrescentar novos comandos em qualquer ponto dela, não apenas no final. O comando `tail` serve para mostrar a parte final de um arquivo ou stream de dados, assim como o seu companheiro `head` serve para mostrar a parte inicial. Para saber mais sobre estes comandos, dê uma olhada em suas páginas de manual.

4. Podemos agora fazer outras coisas como, por exemplo, contar quantos alunos têm uma letra "a" no nome. Basta filtrar o stream de dados com um segundo comando `grep` e usar o comando de contagem `wc`,

```
grep ':3[0-9][0-9][0-9][0-9]:' /etc/passwd \
    | tail +2 | cut -d : -f 5 \
    | grep -i a | wc -l
```

   onde a opção `-i` do comando `grep` o instrui a ignorar a diferença entre letras maiúsculas e minúsculas e a opção `-l` do comando `wc` o instrui a contar linhas. Experimente fazer este exemplo um passo de cada vez, acrescentando cada comando individualmente de forma a poder ver o que acontece nos passos intermediários. Conte quantos alunos do socrates têm cada uma das vogais "a", "e", "i", "o" e "u" em seus nomes. Determine também o número total de alunos que já são usuários do sistema.

5. Para ilustrar como as pipelines de comandos não estão limitadas a operações internas de um determinado sistema, podemos dar um exemplo envolvendo operações através da rede. Vamos fazer no sistema socrates uma filtragem usando `ruptime` de forma a listar o status de todas as máquinas da sala Pró-Aluno. Inicialmente simplesmente execute `ruptime` no socrates para ver o que acontece. Em seguida tente filtrar as máquinas da sala Pró-Aluno usando a pipeline

```
ruptime | grep pop
```

   Finalmente, tente filtrar apenas os terminais da sala usando

```
ruptime | grep 'pop[0-9][0-9]' | tail +2
```

   Construa esta última versão acrescentando uma parte de cada vez.

6. O caracter ; funciona como separador entre comandos, denotando o término do comando anterior, exatamente como acontece na linguagem de programação C. Podemos usar esta sintaxe para executar dois ou mais comandos seguidos em uma única linha, como por exemplo em

   ```
   cd ; ls
   ```

   que nos levará para casa e mostrará o nosso diretório de home. Se queremos realizar alguma operação para a qual seja necessário ou conveniente estar momentaneamente em algum outro diretório, voltando imediatamente para onde estamos agora, podemos usar uma sequência de comandos como, por exemplo, a que segue.

   ```
   cd /usr/local ; echo $cwd ; /bin/pwd ; cd -
   ```

   Outro exemplo seria a sequência de comandos necessários para imprimir um documento `doc.tex` escrito em LaTeX em uma impressora PS. Copie algum arquivo curto de LaTeX já existente com o nome `doc.tex` e execute o exemplo que segue. O comando de impressão vai falhar se não houver uma impressora default funcionando em seu sistema, neste caso troque-o por `gv doc.ps` e tente de novo.

   ```
   latex doc.tex ; dvips doc.dvi ; lpr doc.ps
   ```

   Sequências de comandos como estas são muito úteis em combinação com o mecanismo de "history" da shell, quando é necessário executar a combinação repetidas vezes ao longo de uma sessão. Uma linha de comandos como esta é registrada no sistemas de "history" da shell e pode ser recuperada completa através das teclas de cursor, ao contrário do que acontece com a mesma sequência de comandos emitidos cada um numa linha.

   Observe que estas sequências de comandos podem se parecer um pouco com pipelines, mas trata-se na realidade de uma estrutura completamente diferente. Numa pipeline os programas envolvidos são executados simultaneamente, com suas entradas e saídas conectadas umas às outras. Numa linha de comandos como as que exemplificamos acima os programas são executados um de cada vez em sequência, com os seus mecanismos de entrada e saída funcionando da forma usual. As duas estruturas apenas se tornam aproximadamente equivalentes quando cada um dos programas de uma sequência escreve todo o seu output para um arquivo que será lido pelo próximo programa, como é o caso do exemplo com o documento em LaTeX.

7. Uma das estruturas de controle da shell é a estrutura condicional `if`, que nos permite executar comandos apenas no caso de uma determinada condição estar satisfeita. Em sua forma mais simples esta estrutura pode ser utilizada como no exemplo que segue.

   ```
   if ( "$user" == "<username>" ) echo "username OK"
   ```

   Neste caso é executado o comando `echo` caso a cláusula lógica entre parêntesis seja verdadeira. A estrutura do comando `if` pode ser estendida para permitir a execução de mais de um comando quando a cláusula lógica é verdadeira, mas isto só é útil em programas escritos em scripts de shell, de forma que deixaremos estas extensões para as aulas sobre scripts. Execute uma linha como esta, colocando explicitamente o seu username onde indicado, e veja o que acontece. Troque o seu username por outra palavra e repita o comando.

   Neste caso a cláusula lógica faz uso do teste de comparação `==`, de forma que ela resulta verdadeira se as duas strings de caracteres que a circundam forem iguais. Existem outros testes de comparação, como o teste `!=`, que resulta verdadeiro se as duas strings forem diferentes, ou os testes `<` (menor) e `>` (maior), que são úteis para comparar strings que representem números inteiros. Troque o operador `==` pelo operador `!=` no exemplo acima e volte a executá-lo. Para se informar sobre o conjunto completo de operadores de comparação, olhe o manual da `tcsh`, executando `man tcsh`. Trata-se de um longo manual, que você provavelmente usará muitas vezes.

8. Observe que usamos aspas duplas para demarcar todas as strings no exemplo acima. Em muitos casos é possível omitir estas aspas, o que em geral não causa problemas para strings que estejam escritas explicitamente, a menos que elas contenham espaços em branco. Por outro lado, é prudente sempre usar as aspas no caso do uso de uma variável, como em `$user`, pois os operadores de comparação requerem que haja uma string de cada lado e, se o valor da variável for a string vazia ou simplesmente espaço em branco, resultará um erro de sintaxe.

   Retire todas as aspas e execute de novo o exemplo do ítem anterior. Em seguida execute `set teste = ""`, troque `$user` por `$teste` dentro da cláusula lógica do `if`, omitindo as aspas, e tente executar o exemplo mais uma vez para ver o que acontece.

9. Outro caso que requer cuidado no uso da estrutura `if` é quando as strings envolvidas podem conter espaços em branco, pois neste caso a omissão das aspas fará com que o conteúdo seja interpretado como duas strings em vez de uma, causando erros de sintaxe. Execute o comando `set teste = "A B"` e em seguida execute mais uma vez o comando `if` omitindo as aspas, para ver o que acontece. Depois tente executar o mesmo comando com todas as aspas como no primeiro caso.

10. Existem também operadores de teste para a existência e tipo de arquivos e diretórios. Por exemplo, o operador `-f` testa a existência de um determinado arquivo comum. Para ver como isto funciona, execute o exemplo que segue.

    ```
    if ( -f /etc/passwd ) echo "arquivo existe"
    ```

    Em seguida mude o nome do arquivo para algum outro que não exista e repita o comando. Para verificar que um determinado diretório existe podemos usar o operador `-d`, como no exemplo que segue.

    ```
    if ( -d /etc ) echo "diretorio existe"
    ```

    Depois de executar este exemplo troque `/etc` por um diretório que não exista ou por um arquivo que não seja um diretório e execute novamente o comando. Para se informar sobre o conjunto completo de operadores deste tipo, consulte as páginas de manual da `tcsh`.

11. Além de podermos definir e atribuir valores a variáveis da shell com o comando `set`, como vimos antes, também é possível usar estas variáveis como variáveis numéricas com valores inteiros. Para isto podemos utilizar o operador `@` no lugar do `set` e uma sintaxe de aritmética inteira parecida com a sintaxe do C. Para ilustrar isto, execute a sequência de comandos do exemplo que segue, no qual os caracteres `___>` representam o prompt da sua shell. Observe os espaços em torno do operador `@` e dos operadores aritméticos, eles são necessários.

    ```
    ___> echo $?test
    ___> set test = 0
    ___> echo $?test
    ___> echo $test
    ___> @ test = $test + 10
    ___> echo $test
    ```

```
___> @ test = $test * 2
___> echo $test
___> @ test = $test / 3
___> echo $test
```

Neste exemplo usamos o operador `$?`, que retorna não o valor da variável e sim `0` caso a variável não esteja definida e `1` se ela estiver definida. Desta forma podemos verificar se uma determinada variável está definida antes de usá-la, evitando erros que podem fazer com que o processamento de nossos comandos seja interrompido pela shell. Observe o truncamento na divisão inteira, que acontece da forma que é usual na aritmética inteira de computadores. No exemplo acima poderíamos definir e inicializar a variável tanto com o comando `set test = 0` quanto com `@ test = 0`, com efeitos idênticos.

Note que os valores das variáveis da shell são sempre strings de caracteres. A única diferença é que, no âmbito do comando `@`, a shell interpreta e manipula estas strings de acordo com a aritmética inteira dos números que eles representam. Não se trata, assim, da verdadeira aritmética inteira no computador e estas não são, portanto, operações particularmente rápidas. Não é uma boa idéia usar este tipo de operação para fazer cálculos numéricos maciços com o computador, trata-se de uma estrutura que, como é típico da shell, enfatiza praticidade e não eficiência de execução.

12. Uma das estruturas de controle mais úteis é a estrutura de loop `while`, que nos permite repetir muitas vezes uma determinada lista de comandos. Quando usada na linha de comando, esta estrutura é iniciada com o comando `while (<clausula lógica>)`, onde a cláusula lógica tem exatamente a mesma estrutura e sintaxe daquelas usadas no comando `if`. A partir deste ponto a shell passará a aceitar a lista de comandos a serem executados em loop, enquanto a cláusula lógica for verdadeira, colocando na tela um prompt especial com o formato `while?`, em lugar do prompt usual. A sequência de comandos será terminada quando você digitar o comando `end`. A partir deste momento o seu loop passa a ser executado. Esta estrutura é muito útil em combinação com a aritmética inteira da shell, que nos permite definir contadores. Experimente com o exemplo muito simples que segue.

```
___> set count = 0
___> while ( $count < 10 )
while? @ count = $count + 1
while? echo $count
while? end
```

Verifique bem e corrija cada linha que digitar, pois não será possível voltar para ela para corrigir erros depois de digitar `[Enter]`. Se você acidentalmente colocar um loop infinito para rodar, basta interrompê-lo com `^C`.

13. Também é possível controlar a execução do loop através de um comando `if` colocado dentro dele. Este comando pode acionar condicionalmente o comando `break`, que termina a execução do loop, como se vê no exemplo que segue, que é equivalente ao anterior.

```
___> set count = 0
___> while ( 1 )
while? @ count = $count + 1
while? if ( $count > 10 ) break
while? echo $count
while? end
```

Observe que a cláusula lógica `(1)` é sempre verdadeira, enquanto `(0)` é sempre falsa. O uso da cláusula `(1)` faz com que o loop deste exemplo seja um loop infinito, de forma que o controle possa ser feito pelo comando `if`.

14. O comando `while` pode ser usado para criar pequenos programas de monitoração que fiquem rodando dentro de um terminal. Para isto é útil usar os comandos `clear`, que limpa a tela do terminal, e `sleep`, que faz com que o processo pause e fique esperando por um determinado tempo. O exemplo que segue pode ser usado para monitorar a atividade dos usuários no sistema, listando quantos deles há e dando alguma informação sobre a sessão de cada um.

```
___> while ( 1 )
while? clear
```

```
while? w
while? sleep 5
while? end
```

Este loop rodará uma vez a cada 5 segundos, que é o tempo de espera determinado pelo comando `sleep`. Você pode terminar o loop com `^C`. Caso você venha a ter a necessidade de monitorar de forma contínua alguma coisa, pode escrever um pequeno programa como este e deixar ele rodando dentro de um terminal `xterm` em algum canto de sua sessão X11.

15. Às vezes é necessário que realizemos operações sobre listas de coisas que não estejam necessariamente organizadas numericamente em uma sequência. Neste caso existe uma outra estrutura de controle com a qual também podemos construir um loop, chamada `foreach`. Por exemplo, suponhamos que queremos contar o número de arquivos em cada um de uma lista de diretórios do sistema, neste caso podemos usar uma estrutura como a do exemplo que segue.

    ```
    ___> foreach dir ( /bin /etc /lib )
    foreach? set num = `\ls -1 $dir | wc -l`
    foreach? echo O diretorio $dir contem $num arquivos
    foreach? end
    ```

    A estrutura de aquisição dos comandos para o caso do `foreach` é parecida com aquela para o caso do `while`, mudando apenas o prompt especial que é colocado pela shell. Neste exemplo a variável `dir` é automaticamente definida e o seu valor é trocado por cada uma das strings da lista que aparece entre parêntesis. Em cada um destes casos o conjunto de comandos é executado. Observe o uso de aspas reversas, que são substituídas pela string que resulta da execução do comando ou pipeline de comandos que elas contêm, como já vimos anteriormente.

16. Para que possamos usar a estrutura `foreach` da forma como ela é mais útil, é necessário primeiro que descubramos como criar de forma automática as listas de objetos sobre as quais queremos atuar, atribuindo-as a uma variável. Como um preliminar de nosso exemplo neste sentido, crie um diretório em sua conta, por exemplo com o nome de `anim` e copie dentro dele todos os arquivos que se encontram em

`http://latt.if.usp.br/fma215/materiais/`

nos links relativos a esta aula. Trata-se de uma pequena animação, como é relatado nos créditos que se encontram no arquivo `README`. Use o comando `file` para verificar que trata-se de 40 arquivos em formato JPEG, cada um com um dos quadros da animação, cujos nomes são números, sem a habitual extensão `.jpg`. Em alguns sistemas (não no Linux, por falar nisto, de forma que tudo isto não passa de um exercício acadêmico para nós) é importante que os nomes dos arquivos tenham a extensão apropriada para que os aplicativos os reconheçam, de forma que vamos escrever um pequeno programa para mudar o nome de todos estes arquivos. Isto já é uma coisa ruim de se fazer, um por um, para 40 arquivos, imagine a situação se você estivesse trabalhando numa animação com várias centenas de quadros.

17. Vamos agora criar uma lista dos arquivos e adquirir esta lista como os valores de uma variável da shell. Como você se lembra de aulas anteriores, as variáveis da `tcsh` podem ter um caráter vetorial, com coleções de valores. Entre no seu diretório `anim` e liste os arquivos com o comando `ls -1`. Isto vai produzir uma lista com todos os nomes dos arquivos, incluindo o arquivo `README`, que obviamente não faz parte da sequência de quadros da animação. Podemos filtrar os arquivos que queremos de várias formas diferentes, uma bem simples é `ls -1 0*`. Utilize este comando e a estrutura de aspas reversas para colocar esta lista em uma variável, como está exemplificado a seguir.

    ```
    set files = `\ls -1 0*`
    ```

    O uso do comando `ls` na forma `\ls` evita que seja usado o alias que existe para ele (para verificar, execute `alias ls`), em vez disso é executado o comando diretamente, sem opções. Desta forma evitamos que sejam incluídos na lista de strings os caracteres de controle associados à colorização do output do `ls`, uma opção que usamos por default em nossos sistemas. Estes caracteres de controle poderiam interferir com a execução de nosso programa.

    Para verificar que a variável `files` foi definida como uma variável vetorial, utilize o operador `$#`, que retorna o número de elementos existentes na variável, executando `echo $#files`. Em seguida examine o conteúdo da variável executando `echo $files`. Cada um dos elementos pode ser examinado separadamente através da execução de `echo`

`$files[1]`, `echo $files[2]`, etc. Temos agora a lista completa associada a uma variável. Se tivermos várias centenas de elementos, isto é obviamente bem mais prático do que digitar todos os nomes individualmente na linha de comando.

18. Tendo a lista de objetos sobre os quais queremos operar, podemos agora mudar os nomes de todos os arquivos, de forma rápida e prática, usando o exemplo que segue.

    ```
    ___> foreach file ( $files )
    foreach? echo -n "$file "
    foreach? mv $file $file.jpg
    foreach? end
    ```

    Observe que incluímos uma reportagem de status dentro do loop, usando o comando `echo` para escrever os nomes dos arquivos conforme o programa vai operando sobre eles. O comando `echo -n` escreve para `stdout` a string que aparece em seu argumento sem colocar um caracter de final de linha depois da string. Com o uso das aspas incluímos no final da string um espaço em branco, de forma a delimitar claramente os objetos que estão sendo escritos. Depois de executar este pequeno programa dentro do diretório `anim`, verifique o resultado com o uso do comando `ls`.

    Tente agora "tocar" a animação em seu computador. Existem duas formas de se fazer isto, com o comando `animate` e com o comando `xanim`. A primeira forma funciona um pouco melhor mas requer mais memória do computador, já a segunda é um pouco mais limitada mas requer menos recursos de memória do sistema. Tente primeiro executar, dentro do seu diretório `anim`, o comando `animate *.jpg`. Tenha paciência, pode demorar um pouco para o programa carregar a animação completa. Se isto falhar ou ficar muito lento, tente executar `xanim *.jpg`. Se nenhum dos dois funcionar satisfatoriamente, tente mudar para uma máquina que tenha mais memória RAM. Todas as máquinas da sala Pró-Aluno têm pelo menos 56 MB de memória RAM disponível, o que deve ser suficiente.

19. Um comando muito útil em pipelines é o editor de texto `sed`. Trata-se de um "stream editor", capaz de editar um fluxo de dados de forma não

interativa, segundo um determinado conjunto de regras de edição, não de um editor que se possa usar de forma interativa. Ele é particularmente importante pois permite que as nossas operações de programação de linha de comando possam modificar o conteúdo de arquivos, bem como nos habilita a lidar com fluxos de texto de formas mais complexas. Por exemplo, já vimos que podemos listar todos os alunos do socrates com o comando

```
grep ':3[0-9][0-9][0-9][0-9]:'  /etc/passwd
```

Podemos agora cortar os usernames e os nomes completos usando

```
grep ':3[0-9][0-9][0-9][0-9]:' /etc/passwd \
    | cut -d : -f 1,5 | less
```

Finalmente, podemos usar o sed para trocar o separador : por caracteres [Tab] e escreva o resultado em um arquivo, usando

```
grep ':3[0-9][0-9][0-9][0-9]:' /etc/passwd \
    | cut -d : -f 1,5 \
    | sed -e 's|:|              |g' >! mypasswd.txt
```

onde o espaço em branco dentro do segundo par de barras verticais do argumento da opção `-e` do `sed` consiste de dois tabs seguidos. produzindo assim uma tabela de usernames e nomes. Para digitar o caracter `[Tab]` na linha de comando da shell, digite primeiro o caracter de "quote" da shell, que é `[Ctrl]-V`, depois digite `[Tab]`.

3 Problemas e Dicas

1. Produza uma lista em ordem alfabética contendo todos os nomes dos usuários do sistema. Utilize o comando `sort`, que serve para colocar linhas em ordem num arquivo ou stream de dados. Dê uma olhada nas suas páginas de manual. O comando `tail` também poderá ser útil.

2. Utilizando vários dos elementos que foram explorados nas tarefas desta apostila, determine a soma de todos os números de usuário dos alunos que já têm contas no sistema. Determine também o valor médio destes números, dentro da precisão permitida pelo truncamento que pode acontecer na aritmética inteira.

3. Escreva um programa na linha de comando para calcular o fatorial de 10. Faça com que ele escreva na tela todos os fatoriais dos números de 1 a 10. Veja se você consegue descobrir qual é o maior número cujo fatorial é passível de cálculo desta forma, por aritmética inteira num computador de 32 bits como os Pentiums que temos disponíveis.

4. Escreva um programa na linha de comando para listar todos os 254 possíveis endereços da subrede `192.168.0.n` onde `n` é um número entre 1 e 254, junto com os correspondentes nomes das máquinas, se elas existirem, ou seja, no caso dos endereços que estiverem em uso. Use um loop do tipo `while` com um contador. Lembre-se de que o comando `host`, com um argumento que seja um dos endereços numéricos da rede, retorna o nome da máquina correspondente, se ela existe. Filtre o output deste comando com os comandos `grep` e `cut` de forma a extrair dele apenas o nome da máquina. O comando `echo -n` será útil para a formatação adequada do output.

5. Usando o comando `ping -c 1 <endereço>`, que envia um único pacote ao endereço discriminado, escreva um programa, no estilo do programa do ítem anterior, que liste os endereços e indique quais deles pertencem a máquinas que estão atualmente no ar e, portanto, respondem ao comando `ping`. A estrutura do loop e da filtragem deverá ser muito parecida com a do caso anterior, com o uso dos comandos `grep` e `cut`. Mais uma vez, para conseguir uma boa formatação do output, o comando `echo -n` deverá ser útil. Observe que este programa levará algum tempo para rodar, devido aos "timeouts" do comando `ping` nos casos em que os endereços tentados não estiverem em uso.

   Caso você queira incluir um caracter `[Tab]`, por motivos de formatação, dentro de uma string delimitada por aspas na linha de comando, você pode fazer isto usando o sistema de "quote" da linha de comando, que é o caracter `^V`. Qualquer caracter, inclusive um "tab", que seja digitado logo após um `^V` não será interpretado de forma especial pela shell, como acontece normalmente com a tecla `[Tab]`, que é interpretada como o comando de completamento. Em vez de ser interpretado, o caracter será inserido na linha de comando, no ponto onde estiver o cursor.

6. Descubra como usar o `sed` para trocar colunas num arquivo, usando o arquivo `/etc/passwd` do socrates como exemplo. Você terá de fazer uso

mais sofisticado de regexps. Em particular, você deve usar a sintaxe \(...\) para marcar sub-expressões dentro do primeiro par de barras verticais do argumento da opção -e do sed, bem como a sintaxe \1, \2, etc para as colocar dentro do segundo par de barras verticais. Como exercício, inverta completamente a ordem dos campos daquele arquivo.

4 Uma Lista de Comandos Relevantes

animate <arquivos> : toca uma animação formada por uma sequência de quadros, cada um num arquivo.

break : termina prematuramente a execução de um loop do tipo while ou do tipo foreach; é um comando built-in da shell.

cut [-d <caracter>] -f <campo> : corta o campo de número <campo> de cada linha do input, onde os campos são separados pelo caracter.

end : marca o final do conjunto de comandos a serem executados num loop do tipo while ou do tipo foreach; é um comando built-in da shell.

foreach var (<lista>) : inicia um loop sobre todos os elementos existentes na lista, fazendo com que a variável var seja igual a um dos elementos de cada vez; é um comando built-in da shell.

head -n [<arquivo>] : mostra as primeiras n linhas de um arquivo.

if (<condição>) <comando> : executa o comando se a condição for verdadeira; é um comando built-in da shell.

sed : um editor de texto não-interativo, que funciona como um filtro agindo em uma corrente (stream) de caracteres; extremamente útil para se fazer edição automatizada de arquivos.

sleep n : pausa o processamento por n segundos.

tail <-n|+n> [<arquivo>] : com -n mostra as últimas n linhas de um arquivo; com +n mostra as linhas do arquivo a partir da n-ésima.

wc -<opção> [<arquivo>] : conta caracteres (opção c), palavras (opção w) ou linhas (opção l) em um arquivo.

while (<condição>) : inicia um loop que será executado enquanto a condição for verdadeira; é um comando built-in da shell.

`xanim <arquivos>` : toca uma animação formada por uma sequência de quadros, cada um num arquivo.