Relatório Geral

Israel Prates

Resumo

Este documento aborda conceitos de programação em shell e apresenta problemas e suas respectivas soluções desenvolvidas em Bash e AWK. O capítulo de fundamentação inclui informações acerca de arquivos, comandos, redirecionamento de entrada e saída, strings, expansões e controle de fluxo. As soluções descritas neste documento empregam ainda conceitos adicionais abordados nas respectivas seções dedicadas. Conceitos adicionais incluem funções, expressões regulares, arquivos temporários, tratamento de sinais, arranjos associativos e processamento de opções da linha de comando.

Sumário

Capítulo 1

Introdução

A partir do UNIX, criado na década de 1970 nos Laboratórios Bell da AT&T, outros sistemas operacionais foram desenvolvidos e evoluíram ao longo dos anos. Estes sistemas, como o Linux, BSD, Solaris, AIX, e macOS, foram desenvolvidos por uma comunidade e seus moderadores, como no caso do Linux, ou por empresas que visam comercializar o software — seja o sistema tido como produto final ou parte de outros produtos. A exemplo deste último caso podemos citar Solaris e macOS, desenvolvidos e comercializados por Oracle e Apple, respectivamente.

Um sistema UNIX é aquele certificado pelo consórcio Open Group, de modo que a aderência aos padrões que caracterizam o UNIX seja garantida e comprovada. Um sistema cujo comportamento é o de um sistema UNIX é dito UNIX-*like* ou UNIX-similar, independente de aderência à especificação ou certificação concedida pelo consórcio acima referido. Por simplicidade, não será feita distinção entre as categorias. Deste modo, todas as referências a um sistema UNIX-similar, de agora em diante, serão escritas apenas como UNIX.

1.1 Definição de Shell

Uma shell em um sistema UNIX é uma interface entre o usuário e o sistema operacional. Ela é responsável por captar a entrada, traduzi-la para o sistema, e reportar a saída produzida como resultado.

1.2 História das Shells em UNIX

A shell padrão do UNIX versão 7, de 1979, é a Bourne Shell, que leva o nome de Steven Bourne, seu criador. Esta versão foi a primeira do UNIX a se popularizar, enquanto a shell em questão ainda é parte das versões mais recentes do sistema. Esta shell, apesar de não ter sofrido grandes mudanças desde a sua criação, inspirou o desenvolvimento de outras desde então.

Com o intuito de ser lançada juntamente com a versão BSD do UNIX, a csh, ou C Shell, foi desenvolvida em 1978. Esta shell possui como principal

característica a similaridade entre seus comandos e a sintaxe de C (linguagem de programação).

Em 1986 foi introduzida a Korn Shell, desenvolvida por David Korn nos laboratórios da AT&T. Esta shell comercial foi a primeira opção relevante a fazer frente ao uso de Bourne e C Shell.

Da shell idealizada por Steven Bourne nasceu o Bash. Desenvolvida em 1988, esta shell foi concebido para ser a opção padrão do projeto GNU, idealizado por Richard Stallman, que visa disponibilizar um sistema UNIX com utilitários livres em vez de suas contra-partes proprietárias. Além disto, há em seu nome um trocadilho na forma de acrônimo: Bourne Again Shell, uma referência à expressão *born again*, do inglês.

As primeiras versões do Bash foram desenvolvidas por Brian Fox, que as manteve até 1993. No início de 1989, Chet Ramey se juntou ao esforço de desenvolvimento, corrigindo bugs e incorporando funcionalidades relevantes para o projeto. Nas versões atuais, o mantenedor do Bash é Chet Ramey.

No início do anos 90 surgiu o zsh, uma shell escrita por Paul Falsted tida como opção para o uso de Bash. Uma das primeiras características a se observar no uso desta shell é o completamento de texto, que estende o comportamento de Bash.

1.3 Organização do Documento

O capítulo 2 trata dos fundamentos comuns ao desenvolvimento das soluções propostas em sala: análise de inventário, análise de matrículas, e análise de firewall. Os capítulos 3, 4 e 5, por sua vez, tratam dos principais aspectos distintivos das soluções e das estratégias empregadas.

Todos os scripts abordados neste documento se encontram disponíveis nos apêndices A, B e C, bem como no repositório `https://gitlab.com/israel.prates/ci320-scripts`.

Capítulo 2

Fundamentos

Este capítulo é destinado a fornecer a base que irá permitir a plena compreensão deste documento por parte do leitor. Conceitos abordados aqui são utilizados nos scripts que solucionam os problemas de análise de inventário, análise de volume de matrículas, e filtragem de registros de firewall.

2.1 Arquivos

Existem quatro categorias de arquivos em um sistema UNIX que merecem atenção no escopo deste documento: arquivos comuns, executáveis, links simbólicos e diretórios. Enquanto arquivos comuns, também chamados arquivos de texto, contém dados em formato legível para humanos, arquivos executáveis podem também conter dados binários.

Links simbólicos são arquivos especiais que apontam para outros arquivos. Ao invés de guardar os dados, um link simbólico contém apenas o caminho para o arquivo que referencia. Com este mecanismo é possível referenciar arquivos em qualquer diretório do sistema de maneira conveniente sem interferir com a organização imposta ao ambiente.

Diretórios, por fim, são arquivos especiais que podem conter outros arquivos, sejam eles comuns, executáveis, links, ou até mesmo outros diretórios. São a base para a organização dos arquivos em um sistema UNIX.

Use o comando `file` para identificar o tipo de um arquivo. Este comando recebe como argumento o nome do arquivo testado.

2.2 Comandos

Um comando pode ser um arquivo executável que a shell é capaz de encontrar e referenciar — chamado comando externo — ou um comando interno (*builtin*), que a shell incorpora em seu código fonte e portanto não precisa ser carregado separadamente para ser executado.

A decisão de se implementar um comando como interno ou externo irá depender de seus requisitos de desempenho, visto que a carga de outros executáveis pode impactar negativamente o tempo de execução. Além disto, deve ser observada a necessidade de o comando interagir com estruturas internas da shell, como o comando `cd`, que muda o diretório corrente, e portanto precisa ser um *builtin* para desempenhar sua função.

Um comando pode ser invocado indicando seu nome e, se desejado ou necessário, seus argumentos. Argumentos são trechos de texto adicionais que seguem o nome do comando e podem indicar o chaveamento de opções, o nome de outros arquivos, ou simplesmente dados a serem processados pelo comando em questão.

Observe que, por serem executáveis, os comandos externos podem estar em uma localidade arbitrária do sistema. Portanto, para que um comando possa ser invocado apenas pelo seu nome ao invés de seu caminho completo, o caminho para o seu diretório precisa ser listado na variável de ambiente `PATH`. Esta variável indica à shell quais diretórios do sistema devem ser verificados para se encontrar o comando requisitado pelo usuário.

Use `type` para identificar se um comando cujo nome seja dado como argumento é interno ou externo. Use `man` para ler o manual de um comando externo dado como argumento, ou `help` para ler a página de ajuda de *builtins*. O comando `help` com a opção `-m` exibe a mensagem de ajuda formatada de maneira semelhante à uma entrada de manual.

2.3 Redirecionamento de Entrada e Saída

A capacidade de se redirecionar a saída de um comando para a entrada de outro é uma ferramenta poderosa para a construção de sistemas de informação complexos, e ainda assim eficientes. Este tipo de redirecionamento é possível através do operador `|`, chamado *pipe*.

O encadeamento de múltiplos programas por meio de *pipes* é chamado *pipeline*. A linha `comando_1 | comando_2` executa `comando_1` e transforma sua saída em entrada para `comando_2`. Esta construção permite o uso de utilitários especializados para o tratamento sucessivo dos dados, viabilizando a abstração dos detalhes das etapas de processamento. Quanto ao número de comandos que podem ser encadeados, este limite irá depender de quantos descritores de arquivo (*file descriptors*) podem ser abertos simultaneamente em seu sistema. Este número é comumente de milhares de descritores[1].

Outro tipo de redirecionamento é o de arquivos. Representado pelos operadores `>` e `<`, este tipo de redirecionamento permite salvar a saída de um comando em um arquivo (operador `>`) ou ler a entrada a partir de um arquivo (`<`). Seu uso pode ser feito isoladamente, como em `comando_1 > arquivo_1`, que redireciona a saída de `comando_1` para `arquivo_1` (de maneira análoga, se `<` tivesse sido usado,

[1] Teste o *builtin* `ulimit` com opções `-n` e `-H` para descobrir o limite de descritores em seu sistema. Leia a página de ajuda do comando para descobrir outras opções.

`comando_1` teria lido a entrada a partir de `arquivo_1`), ou em conjunto com outro redirecionamento ou pipeline: `comando_1 < arquivo_1 | comando_2 > arquivo_2`[2].

Estes recursos, quando associados à filosofia de se construir utilitários especializados e com grande número de otimizações, são parte central no processo de viabilização de soluções complexas, eficientes e desenvolvidas em tempo hábil em UNIX.

2.4 Strings

Uma string, ou cadeia de caracteres, pode ser representada em Bash por qualquer sequência de caracteres entre aspas do mesmo tipo (`" "` ou `' '`). Caso sejam utilizadas as aspas duplas, alguns caracteres ainda serão interpretados como especiais, enquanto ao utilizar aspas simples nenhum caractere especial reterá suas propriedades. Um uso para este comportamento é o de permitir ou não a expansão de variáveis ou comandos em strings: `echo "$my_var"` irá imprimir o conteúdo da variável `my_var`, enquanto `echo '$my_var'` irá imprimir o texto `$my_var`.

2.5 Expansão

Expansão pode se referir a três conceitos distintos em Bash: expansão de caminho, também conhecida como *globbing*, expansão de chaves, e expansão de variáveis.

2.5.1 Expansão de Caminho

A expansão de caminhos é a capacidade de, por meio de padrões, especificar o nome de arquivos sem conhecê-los de antemão. Este tipo de expansão, portanto, depende do conceito de coringas: construções capazes de representar uma classe ou conjunto de caracteres. A seguir os principais coringas em Bash são listados:

- `?`: Qualquer caractere único.
- `*`: Quaisquer caracteres, independente de quantidade.
- `[characters]`: Quaisquer caracteres listados em *characters*.

Algumas observações acerca de coringas:

1. As expansões de metacaracteres ocorrem no nível da shell, fora do domínio de quaisquer programas nos quais elas sejam usadas para definir argumentos.
2. Todos os coringas podem ser usados em conjunto com radicais fixos. Quando utilizados desta forma, deve haver correspondência entre o nome do arquivo expandido e o radical para que a expansão ocorra.

[2] Qual o fluxo de dados para este encadeamento?

3. Todos os coringas podem ser combinados e repetidos em uma expressão de expansão de caminho.

4. O metacaractere `*` permite expansões vazias. Isto implica que, se necessário, `*` será expandido em uma cadeia vazia para que a expansão ocorra.

5. O operador de conjunto (caracteres `[` e `]`) faz com que qualquer um dos caracteres entre `[` e `]` seja expandido uma vez.

6. O operador de conjunto permite negar o seu conteúdo, fazendo com que apenas caracteres que não estão listados sejam expandidos. Isto pode ser feito ao se adicionar `!` imediatamente após `[`, como em `[!123]`, que expande todo caractere que não seja `1`, `2` ou `3`.

7. O operador de conjunto suporta intervalos, como em `[a-j]`, onde qualquer caractere entre `a` e `j`, em ordem lexicográfica, será expandido.

8. Para listar um hífen em um conjunto basta fazê-lo por último, como em `[a-c1-3-]`. Este conjunto permite o reconhecimento dos caracteres `a`, `b`, `c`, `1`, `2`, `3`, e `-`.

9. O operador de conjunto permite usar negação e intervalos na mesma expressão.

10. Caso os metacaracteres não possam ser expandidos, a cadeia que representa o padrão permanecerá intocada e será dada como argumento para o comando sendo chamado.

2.5.2 Expansão de Chaves

A expansão de chaves permite expandir um radical e duas ou mais variantes. Pode ser invocado como em `touch sort.{c,h}`, onde `sort.` é o radical, e `c` e `h` as variantes. O resultado desta chamada é equivalente a `touch sort.c sort.h`.

A expansão de chaves pode ser repetida: `touch {a,b}x{1,2}`, que equivale a `touch ax1 ax2 bx1 bx2`.

Além disto, a expansão de chaves pode ser aninhada, como em `touch aaa{bbb,c{c,d,e}c,fff}ggg`, equivalente a `touch aaabbbggg aaacccggg aaacdcggg aaacecggg aaafffggg`.

Vale observar que a expansão de chaves não depende de um nome existente no caminho correspondente para ocorrer. Este tipo de expansão pode, ainda, ser combinado com a expansão de caminho.

2.5.3 Expansão de Variáveis

Uma variável em Bash pode ser expandida utilizando o caractere especial `$`. Este caractere seguido de um identificador causa a expansão da variável, como na linha a seguir. Observe que o uso de chaves é opcional para a maioria dos casos de expansão, mas ainda assim uma boa prática a ser seguida.

```
1 echo ${my_variable}
```

Scripts e funções em Bash são capazes de receber parâmetros, que por sua vez são expandidos como variáveis que tem como identificadores suas posições. O trecho de código abaixo expande os parâmetros passados nas três primeiras posições. Observe que em um script a posição zero se refere ao nome do arquivo que o contém.

```
echo ${1} ${2} ${3}
```

Para se expandir um parâmetro posicional entre 0 e 9 o uso de chaves é opcional, porém para as posições a partir da décima seu uso se torna obrigatório.

Existem ainda outras formas de expansão que, ao serem usadas, propiciam maior controle sobre as variáveis. Note que em todas as formas listadas a seguir o uso de chaves é obrigatório.

Valor Padrão

Bash provê um mecanismo que permite a expansão de um valor padrão em duas condições: quando um dado identificador não estiver definido ou quando o valor da variável for nulo. Este comportamento é obtido quando o identificador é sucedido pela construção `:-value`, onde `value` é o valor que se deseja tomar como padrão. Note que `value` não está restrito a constantes e é sujeito a outras expansões, como de caminho ou variável.

```
declare my_var='' # Declara identificador my_var com valor nulo.
echo ${my_var:-123} # Escreve '123'.
unset my_var # Remove my_var.
echo ${my_var:-123} # Escreve '123'.
```

Ao omitir o caractere dois pontos (:) da expansão, o comportamento obtido é o de expandir o valor padrão apenas em casos de identificador não definido. Deste modo, se a variável estiver definida e seu valor for nulo, ainda assim seu valor será expandido ao invés do valor padrão.

```
declare my_var='' # Declara identificador my_var com valor nulo.
echo ${my_var-abc} # Escreve ''.
unset my_var # Remove my_var.
echo ${my_var-abc} # Escreve 'abc'.
```

Caso se deseje atribuir um valor à variável antes de se executar a expansão, o símbolo de atribuição (caractere =) deve ser utilizado ao invés do traço, como a seguir. Note que a omissão dos dois pontos produz, neste cenário, efeito análogo ao que produzia no anterior.

```
unset my_var # Remove my_var, caso exista.
echo ${my_var:=abc} # Atribui 'abc' a my_var. Em seguida expande my_var.
```

Expansão de Substring

Dada uma cadeia de caracteres, uma substring é um trecho contíguo de caracteres desta cadeia, onde o comprimento do trecho é limitado pelo comprimento da cadeia que o contém. Uma substring pode ser obtida em Bash através de expansões do tipo `identificador:inicio:comprimento`, onde `inicio` indica quantos caracteres a contar do primeiro devem ser ignorados no valor referenciado por `identificador`.

A componente `:comprimento` é opcional e indica o número máximo de caracteres a serem expandidos.

```
declare my_var='abcdefghi' # Atribui a string 'abcdefghi' a my_var.
echo ${my_var:3} # Escreve 'defghi'.
echo ${my_var:0:3} # Escreve 'abc'.
echo ${my_var:3:3} # Escreve 'def'.
echo ${my_var:6:3} # Escreve 'ghi'.
```

Eliminação de Padrão

Dado um padrão que respeite as regras de expansão de caminho, é possível eliminar a menor ocorrência deste padrão a contar do início de uma string. A sintaxe para este tipo de expansão é `identificador#padrao`. De modo a forçar a eliminação da maior ocorrência possível o separador entre o identificador e o padrão deve ser `##`, de modo a produzir construções da forma `identificador##padrao`.

```
declare my_var='abc_def_ghi' # Atribui a string 'abc_def_ghi' a my_var.
echo ${my_var#*_} # Escreve 'def_ghi'.
echo ${my_var##*_} # Escreve 'ghi'.
```

De maneira análoga, o caractere % permite o casamento do padrão a contar do fim de uma string.

```
declare my_var='abc_def_ghi' # Atribui a string 'abc_def_ghi' a my_var.
echo ${my_var%_*} # Escreve 'abc_def'.
echo ${my_var%%_*} # Escreve 'abc'.
```

Substituição de Padrão

Dado um padrão que respeite as regras de expansão de caminho, a construção `identificador/padrao/constante` irá substituir a primeira ocorrência do padrão pela constante indicada. Caso o comportamento desejado seja o de continuar tentando substituir mesmo após a primeira ocorrência, duas barras devem ser usadas para separar o identificador do padrão, resultando em `identificador//padrao/constante`.

```
declare my_var='abcabcabc' # Atribui a string 'abcdefghi' a my_var.
echo ${my_var/abc/xyz} # Escreve 'xyzabcabc'.
echo ${my_var//abc/xyz} # Escreve 'xyzxyzxyz'.
echo ${my_var/abc*/xyz} # Escreve 'xyz'.
echo ${my_var//abc*/xyz} # Escreve 'xyz'.
```

Observe que a constante pode ser omitida, resultando em remoção do padrão em vez de substituição. Neste caso, o caractere / que separa o padrão e a constante também se torna opcional.

```
declare my_var='abcabcabc' # Atribui a string 'abcdefghi' a my_var.
echo ${my_var/abc/} # Escreve 'abcabc'.
echo ${my_var/abc} # Escreve 'abcabc'.
echo ${my_var//abc} # Escreve ''.
```

2.5.4 Substituição de Comando

A saída de um comando pode ser expandida de maneira semelhante à de uma variável por meio de construções da forma `$(comando)`. Este mecanismo é particularmente interessante para atribuição de variáveis e formatação de strings, como nos exemplos a seguir.

```
file_name=$(basename full_path) # Salva em file_name apenas o nome do arquivo.
echo "Sua pasta atual: $(pwd)" # Expande o resultado do comando pwd na string.
for item in $(ls *.txt); do # Lista arquivos com extensao .txt.
    echo "-${item}-" # Imprime o nome do arquivo entre tracos.
done
```

Observe que a substituição de comando pode ser aninhada e conter expansões dos outros tipos. Além disto, redirecionamento de entrada e saída também é permitido neste tipo de construção.

2.6 Controle de Fluxo

É possível controlar o fluxo de execução em um script por meio das palavras-chave `if`, `while` e `for`. Estas estruturas modificam o fluxo de execução do programa e permitem o desvio condicional de operações (`if`) ou repetição de comandos (`while` e `for`).

O desvio condicional em sua forma mais simples pode ser descrito como `if condicao; then comando_1; comando_2; ... ; comando_n; fi`, onde `condicao` é um comando qualquer ao qual o bloco de comandos de 1 a n tem sua execução condicionada. Caso o comando `condicao` retorne com sucesso o bloco é executado, caso contrário não.

O laço de repetição `while`, da forma `while condicao; do comando_1; comando_2; ... ; comando_n; done`, irá executar, repetidamente, os comandos de 1 a n enquanto o comando representado por `condicao` retornar com sucesso.

O laço `for` permite iterar por itens de uma coleção, e é da forma `for item in colecao; do comando_1; comando_2; ... ; comando_n; done`. A variável `item` irá assumir a cada iteração o valor de um item de `colecao`, até que a coleção se esgote. Para cada iteração o bloco de comandos indicado será utilizado.

Capítulo 3

Análise de Inventário

Neste capítulo é apresentada a implementação em Bash de uma solução que permite a análise do inventário do departamento de informática na Universidade Federal do Paraná. Aspectos relevantes deste problema são o uso de expressões regulares e separação de strings para a filtragem do arquivo de entrada, declaração de variáveis, declaração de funções, e criação de arquivos temporários.

3.1 Descrição do Problema

Esta solução visa fornecer os meios para exibir, de maneira organizada, o patrimônio do departamento ao qual o autor pertence. Os dados de entrada são fornecidos em um único arquivo de texto no formato CSV, ou *Comma-separated Values*, onde cada campo é uma cadeia de caracteres entre aspas duplas. O caractere que separa os campos é o ponto e vírgula (;). Ainda, é de se observar que, caso existam linhas cujos campos contenham aspas duplas ou ponto e vírgulas dentro de ao menos uma cadeia de caracteres, a linha inteira deve ser desconsiderada.

Por fim, esta solução deve produzir como saída um conjunto de arquivos onde o nome de cada um é o campo de localização (coluna 5 do arquivo CSV). Neles devem estar contidas as linhas que correspondem à localização em questão.

3.2 Implementação em Bash

Para usar esta solução o arquivo de entrada deve estar na mesma pasta que o script `inventory.sh`, e deve se chamar `file-01.csv`. O script `inventory-fetch-source.sh` foi criado para automatizar esta parte do processo. Executar este script fará com que os dados de entrada sejam copiados da pasta pessoal do usuário `marcos` no domínio `ssh.inf.ufpr.br`, autenticado por `ifp15`, para o caminho relativo `./file-01.csv`.

3.2.1 Estratégia

A estratégia adotada foi primeiramente filtrar o arquivo de entrada, desconsiderando as linhas com aspas duplas ou ponto e vírgulas proibidos. Vencida esta etapa, apenas os valores da quinta coluna (as localizações) foram salvos em um arquivo temporário, de maneira única e ordenada. Por fim, munido deste arquivo temporário, um novo arquivo foi criado para cada localização, e preenchido com linhas do arquivo de entrada tal que o campo de localização corresponde ao nome do arquivo.

O restante deste capítulo se dedica a detalhar os conceitos empregados para a realização desta abordagem.

Declaração de Variáveis e Funções

Uma variável em bash é um identificador que representa uma área de memória que o interpretador é capaz de referenciar. O comando interno `declare` permite a declaração de variáveis. Em sua forma mais simples este comando recebe como argumento um identificador e reserva o nome indicado, como abaixo.

```
1 declare my_variable
```

A variável declarada pode, opcionalmente, ser inicializada, como na linha a seguir.

```
1 declare -r source='./file-01.csv'
```

Observe a opção `-r`. Esta opção reserva o nome como sendo somente leitura, de modo que escritas subsequentes à inicialização da variável não são possíveis. Leia a página de ajuda do comando `declare` para descobrir outras opções.

Funções em bash podem ser declaradas através do *builtin* `function`, seguido do nome e do corpo da função. O corpo é um bloco de comandos que será executado cada vez que a função for invocada.

```
1 function clean {
2     rm -f "$@"
3 }
```

Uma função também pode ser declarada informando apenas seu nome, abre e fecha parênteses e o corpo (sem o *builtin* `function`).

```
1 clean() {
2     rm -f "$@"
3 }
```

Funções podem receber e declarar argumentos argumentos da mesma forma que

Expressões Regulares e Comando *grep*

Está fora do escopo deste documento explicar expressões regulares e autômatos finitos em detalhe. No entanto, o comando `grep` e a expressão usada para filtrar o documento estão. O comando `grep` recebe como argumento uma expressão regular na forma de uma string e irá casar a expressão com o texto lido da

entrada padrão. Este comando, em sua forma mais simples, irá produzir como saída apenas as linhas onde a expressão foi reconhecida.

A expressão usada para a filtragem do arquivo exclui — ou seja, é incapaz de reconhecer — linhas que contenham o caractere aspas duplas (") ou ponto e vírgula (;) entre aspas duplas seguidas por um ponto e vírgula. A expressão pode ser vista no trecho de código abaixo.

```
declare -r pattern='^(("[^";]*")+;)+$'
```

Se descrita de maneira verbal, esta expressão equivale a dizer: execute a leitura a seguir uma ou mais vezes. Execute a leitura a seguir uma ou mais vezes e depois leia um ponto e vírgula. Leia aspas duplas, seguidas por qualquer quantidade de caracteres que não sejam aspas duplas ou ponto e vírgula, seguido de aspas duplas.

Comando *cut*

O comando externo `cut` recebe como argumento opcional um caminho de arquivo. Caso não seja fornecido um caminho, então este comando lerá a entrada padrão. Este comando permite escrever apenas parte do que lhe for dado como entrada, utilizando como mecanismo de filtragem um `offset` em bytes ou caracteres. Além desta, outra opção de filtragem é o uso de campos e separadores, onde um caractere separador é indicado e um campo é referenciado pelo seu número.

A linha abaixo lê a entrada a partir do arquivo cujo nome está na variável `filtered`, indica o caractere ponto e vírgula como separador, e então filtra cada linha do arquivo para exibir apenas o quinto campo. A saída deste comando é ligada à entrada do utilitário `sort`, com opção `-u`, que retorna os elementos únicos em ordem lexicográfica.

```
cut -d';' -f5 "${filtered}" | sort -u > "${locations}"
```

Arquivos Temporários

Para criar os arquivos temporários foi utilizado o comando externo `mktemp`. Este comando recebe como parâmetro opcional uma string que representa o padrão que irá determinar o nome de um arquivo a ser criado. Se o padrão for suprido, os caracteres `'X'` são substituídos por caracteres aleatórios no nome do arquivo. Se não for fornecido um padrão, então um arquivo com nome aleatório é criado na pasta de arquivos temporários do seu sistema. Este comando escreve o caminho do arquivo recém criado na saída padrão.

Para impedir que os arquivos temporários poluam o sistema, foram utilizados o *builtin* `trap` e a função `clean`, definida pelo autor. A função recebe como argumentos uma sequência de nomes de arquivos, e os remove sem produzir saída. O comando `trap`, por outro lado, recebe como argumento um bloco de texto e um indicador de sinal do sistema. Quando o sinal indicado em `trap` for recebido, o bloco de texto associado é executado. A chamada abaixo associa a linha `clean ${temp_semesters[@]}` ao sinal `EXIT`, que é recebido mediante saída (prematura ou não) do programa.

```
1 trap 'clean ${temp_semesters[@]}' EXIT
```

Note que, como aspas simples foram utilizadas, os elementos do arranjo serão expandidos apenas quando a chamada a `clean` ocorrer[1].

[1] O que aconteceria se aspas duplas fossem utilizadas? Se entre a chamada do comando `trap` e a saída do programa o arranjo fosse alterado, o que a função receberia?

Capítulo 4

Volume de Matrículas

Neste capítulo é apresentada a implementação em Bash de uma solução que permite a análise do volume de matrículas em disciplinas do departamento de informática na Universidade Federal do Paraná. O período analisado está contido entre o primeiro semestre de 1988 e o segundo semestre de 2002. Entre as técnicas empregadas para a elaboração da solução estão uso de arranjos associativos, navegação na árvore de diretórios usando `pushd` e `popd`, bem como processamento de opções na forma de argumentos provenientes da linha de comando.

4.1 Descrição do Problema

O problema aqui abordado é o de discernir a evolução do volume de matrículas ao longo dos anos. Um sistema de informação capaz de resolver este problema deve receber como entrada dados históricos do período em análise e produzir como saída dois relatórios: um simplificado, que reporta apenas a quantidade de matrículas por disciplina, e outro detalhado, capaz de discriminar estes dados por semestre.

Os dados de entrada são fornecidos por meio de arquivos compostos por duas colunas separadas por dois pontos (:). As colunas, por sua vez, representam os campos curso e grr, conforme evidenciado no cabeçalho presente em todos os arquivos. Com isto, podemos extrair informações importantes sobre a estrutura dos documentos, como o número mínimo de linhas presente em cada arquivo, qual o caractere separador e o número esperado de colunas.

Cada arquivo pertencente à classe descrita anteriormente é uma listagem de matrículas por semestre em uma dada disciplina. Sabemos ainda que seus nomes são compostos pelo ano a que se referem (entre 1988 e 2002), seguido do número do semestre (1 ou 2), com a extensão `.dat`[1]. Os dados históricos de cada disciplina são representados por arquivos desta classe, organizados em diretórios por disciplina. Por fim, todas as matrículas conhecidas e relevantes para este

[1] A extensão originalmente adotada é `.dados`, no entanto, no script apresentado no anexo B.1, esta extensão foi alterada para `.dat`

problema são fornecidas por meio deste esquema, de modo que se assume que todas as premissas aqui descritas são verdade.

4.2 Implementação em Bash

Para usar esta solução você deve, primeiro, executar o script `registration-fetch-source.sh` para obter os arquivos de entrada no formato esperado pelos demais scripts. Isto fará com que os dados de entrada sejam copiados do caminho em `source` para `destination`, e então extraídos na pasta especificada pela variável `extracted`. Um link simbólico, cujo caminho é dado por `link_name`, é criado para facilitar a referência aos arquivos.

A solução consiste em dois casos de uso distintos, representados pelos scripts `registration-simple.sh` e `registration-detailed.sh`, detalhados nas seções que seguem.

4.2.1 Estratégia da Análise Simplificada

No caso simplificado o objetivo é reportar a quantidade de matrículas por semestre em todas as disciplinas, sem distinção. Alunos repetidos não devem ser computados múltiplas vezes neste caso. A estratégia adotada foi a de agrupar os alunos matriculados por semestre em arquivos temporários. Estes arquivos são nomeados de acordo com o semestre a que se referem, de modo que ao computar a quantidade de linhas únicas em cada arquivo se obtém a contagem de alunos únicos matriculados por semestre.

Arranjos Associativos

No contexto desta abordagem, os arranjos associativos são a principal estrutura de dados, visto que permitem agrupar os dados dos semestres por uma chave em comum: o próprio semestre. Assim, se torna viável a computação das matrículas únicas para cada semestre pois, ao forçar a colisão de chaves, o número de matrículas únicas pode facilmente ser obtido.

Um arranjo associativo é um arranjo capaz de aceitar strings como índices e, em Bash, pode ser declarado através do *builtin* `declare` com opção `-A` como a seguir.

```
1 declare -A temp_semesters=()
```

Aqui o arranjo associativo `temp_semesters` é criado e inicializado como vazio. O arranjo pode também ser inicializado com valores arbitrários, como na chamada abaixo.

```
1 declare -A my_array=([key_1]=value_1 [key_2]=value_2)
```

O valor associado a um índice presente no dicionário pode ser referenciado ou atribuído utilizado o operador de acesso a índices `[ ]`, como a seguir, onde o arranjo `temp_semesters` recebe o valor armazenado na variável `temp` no índice cujo valor é dado por `semester`.

```
1 temp_semesters["${semester}"]="${temp}"
```

Expandir todos os valores do arranjo pode ser feito por meio do uso dos caracteres `@` e `*` como índices, como em `${my_array[@]}` e `${my_array[*]}`. Usar o caractere `@` faz com que cada elemento do arranjo seja expandido como uma palavra distinta da shell, enquanto o uso de `*` faz com que a expansão resulte em uma única palavra tal que os elementos são separados pelo primeiro caractere da variável de ambiente `IFS` (*Internal Field Separator*).

Para se expandir todas as chaves do arranjo é necessário inserir o caractere `!` imediatamente antes do identificador do arranjo, como em `echo "${!my_var[@]}"`, que resulta na expansão de todas as chaves de `my_var`.

4.2.2 Estratégia da Análise Detalhada

No caso detalhado o objetivo é reportar a quantidade de matrículas por semestre discriminando disciplinas. Alunos que tenham se matriculado em múltiplas disciplinas devem ser computados para cada matrícula.

A estratégia adotada foi de se navegar pela estrutura de diretórios — representando uma disciplina — e computar a quantidade de matrículas para cada semestre, o que resulta em uma matriz de quantidades de matrículas onde cada linha é uma disciplina e cada coluna é um semestre. Sabendo que cada arquivo de semestre contém ao menos uma linha referente ao cabeçalho, o número de matrículas é dado pelo número de linhas do arquivo subtraído em um.

Navegação na Árvore de Diretórios

Visto que parte central desta estratégia é a navegação na estrutura de diretórios, esta parte do documento trata de como fazê-lo por meio de `pushd` e `popd`.

Os comandos `pushd` e `popd` são `builtins` que permitem a navegação na estrutura de diretórios do sistema por meio de uma pilha. Assim, o comando `pushd` empurra um novo diretório no topo da pilha, enquanto o comando `popd` é a operação análoga de remoção do topo. Estes comandos são valiosos em cenários onde a navegação em estruturas complexas se faz necessária. Eles permitem subir e descer na árvore de diretórios, ao mesmo tempo que mantém um registro do caminho da raiz até o nó corrente.

O comando `pushd` recebe como argumento o nome do diretório que será colocado no topo da pilha, faz com que o novo topo se torne o diretório corrente, e produz como saída o estado da pilha após a inserção. O comando `popd` não recebe argumento, remove o diretório no topo da pilha, transforma o novo topo em diretório corrente, e produz como saída o estado da pilha após a remoção.

Processamento de Opções

Esta solução permite chavear a exibição do cabeçalho e a formatação das colunas por meio de opções provenientes da linha de comando. Tendo isto em mente, segue uma breve revisão do mecanismo utilizado para o reconhecimento das opções.

Uma forma de se processar opções dadas a um script em Bash é por meio do comando externo `getopt`. Este comando permite ler os argumentos de um script de modo a reconhecer um conjunto de opções previamente listadas. Caso seja fornecida uma opção desconhecida o comando retorna com saída anormal.

O comando `getopt` permite definir opções curtas da forma `-c` onde `c` é um caractere único que representa a opção. A opção `-o` (ou `--options`) deve ser utilizada para que opções curtas sejam especificadas. Esta opção recebe a lista de caracteres reconhecidos, como no exemplo a seguir, onde as opções `-h` e `-p` são listadas como conhecidas.

O comando também permite o reconhecimento de opções longas. Estas opções, da forma `--string` podem ser reconhecidas pela opção `-l` (ou `--long`), que por sua vez recebe a lista de strings que devem ser reconhecidas separadas por vírgulas. O exemplo abaixo reconhece as opções `--header` e `--pretty-print`.

A opção `-n` (ou `--name`) indica ao comando `getopt` o nome do programa que está sendo executado. Esta informação é dada para fins de formatação de mensagens de erro.

```
1 options=$(getopt --options hp --long header,pretty-print --name "${0}" -- "$@")
2 [ "$?" != 0 ] && exit 1 # O caractere ? expande o retorno do comando anterior.
```

Observe que a expansão dos argumentos do script ocorre no trecho `"$@"`, onde os argumentos são dados a `getopt`. Caso os argumentos não sejam reconhecidos, o comando irá falhar e causar a saída prematura na linha que segue.

Capítulo 5

Análise de Requisições Filtradas

Neste capítulo é apresentada a implementação em AWK e Bash de uma solução que permite a análise do volume de requisições filtradas no firewall do departamento de informática na Universidade Federal do Paraná. Este programa deve produzir como saída um sumário do volume de requisições agrupado pelo nome da regra filtrada. Apesar de ser uma tarefa simples, o desempenho é crítico, de modo que o tempo de execução do filtro é limitado a 0.5 segundo para entradas grandes (de tamanho superior a 100 MB). A principal contribuição é a introdução à linguagem AWK.

5.1 Descrição do Problema

O problema aqui abordado é o de sumarizar as requisições por tipo e reportar a quantidade de ocorrências. Caso um número anormal de requisições (20000) seja atingido, uma mensagem de notificação deve ser disparada. Para os dados de entrada disponibilizados, nenhuma execução do filtro pode ultrapassar 0.5 segundo. Para atingir este requisito de desempenho foi empregada a linguagem AWK, uma linguagem específica para processamento de texto.

Os dados de entrada são fornecidos por meio de um único arquivo composto por colunas separadas por espaços. A sexta coluna é a única relevante para a solução deste problema, pois contém o nome da regra que causou a filtragem da requisição.

5.2 Implementação em AWK e Bash

Para usar esta solução basta chamar o script `firewall.awk`. Este script lê os dados da entrada padrão e produz como saída o sumário das regras encontradas no registro de firewall. Em sua execução o script irá, de maneira automática, execu-

tar um outro arquivo (escrito em Bash) responsável por enviar uma mensagem em casos onde o número de ocorrências é superior a 20000. Caso este comportamento de envio seja indesejado, você deve fornecer a opção `-v suppress_email=true` ou `-v nomsg=true`. Estas opções irão definir as variáveis `suppress_email` ou `nomsg`, respectivamente, como verdadeiras. Ao fazer isto nenhuma mensagem será enviada.

5.2.1 Estratégia

A abordagem adotada é centrada em arranjos, que em AWK são sempre do tipo associativo. Ao indexar o arranjo usando o nome da regra associada à ocorrência é possível manter um contador para cada regra distinta. A listagem de regras possíveis é lida ao início de cada execução, e é utilizada para inicializar os contadores. Antes de se encerrar a execução, o número de ocorrências é verificado e, caso se exceda o limite de 20000 ocorrências, um email de notificação é disparado.

Campos

Campos, também chamados colunas, podem ser referenciados de maneira simples através do uso do símbolo `$` e um número, como em `$6`, onde o sexto campo de uma dada linha é referenciado. Visto que apenas este campo é interessante ao domínio da aplicação, este mecanismo provê uma maneira direta de se acessar a informação relevante. Por padrão, o caractere espaço age como separador de campos.

Variáveis

Variáveis não são fortemente tipadas e podem ser definidas informando o identificador seguido do símbolo de atribuição (`=`) e o valor que será atribuído, como em `my_var="abc"`. Em arranjos a inserção de um novo elemento é feita ao atribuir um valor a um endereço (índice) não ocupado.

Blocos

Blocos são trechos de código agrupados por chaves e podem ser dos tipos principal, `BEGIN` e `END`. É no bloco principal onde as ações que serão executadas para cada linha do arquivo são definidas. Os dois blocos restantes, `BEGIN` e `END`, são opcionais. Se definidos, serão executados antes e depois do bloco principal, respectivamente.

Um programa simples que imprime um cabeçalho, um rodapé, e filtra apenas a primeira coluna da entrada do programa pode ser visto a seguir. Observe que o bloco principal não possui um identificador como os blocos `BEGIN` e `END`.

```
1 BEGIN {
2     print "###"
3 }
4 {
5     print $1
6 }
```

```
END {
    print "###"
}
```

Na solução proposta os três blocos foram utilizados. No bloco `BEGIN` é feita a leitura das possíveis regras do firewall a partir de um arquivo e a inicialização dos contadores de ocorrência. No bloco principal o sexto campo de cada linha lida é utilizado para acessar os contadores de modo a incrementá-los. Por fim, no bloco `END` os contadores e suas chaves são escritos na saída padrão.

É também no bloco `END` que as mensagens de notificação são enviadas — apenas em caso de uma regra ter número de ocorrências maior ou igual a 20000. Esta funcionalidade é implementada através da função `system`, que recebe uma string como argumento. Esta string é repassada para a shell, que por sua vez a interpreta como um comando. A função `system` retorna o código de retorno do comando executado na shell. Deste modo é possível executar um script em Bash a partir de um programa em AWK.

Here Documents (Bash)

A construção chamada *here documents* é utilizada no script responsável pelo envio das mensagens. Este tipo especial de redirecionamento permite indicar uma lista de comandos ou constantes que será processada e então redirecionada para outro comando, de forma semelhante como seria um arquivo. O trecho de código a seguir demonstra esta construção. Observe que as variáveis e comandos são expandidos normalmente mesmo neste cenário.

```
cat << FIM_DE_ARQUIVO
Nome: $(whoami)
Pasta pessoal: $HOME
Root? $([ "$(id -u)" == "0" ] && echo 'S' || echo 'N')
FIM_DE_ARQUIVO
```

Um delimitador de fim de arquivo deve ser informado imediatamente após o símbolo de redirecionamento. Este delimitador pode ser qualquer sequência de caracteres e irá produzir o fim do bloco apenas quando encontrado sozinho em uma linha. No exemplo a seguir a linha `AOF BOF COF DOF EOF` é impressa, apesar de conter o delimitador, pois este veio acompanhado de outros caracteres.

```
cat << EOF
AOF BOF COF DOF EOF
EOF
```

Capítulo 6

Conclusão

Desde a história das shells acompanhada de um vislumbre da filosofia UNIX; passando pelas aplicações que ilustraram os conceitos investigados aqui — aplicações não apenas ilustrativas, mas viáveis para solucionar problemas reais do departamento; chegando, por fim, aos pormenores das expansões de variáveis e usos criativos de arranjos associativos: o conteúdo para o qual este documento foi proposto, de minha parte, foi coberto.

Áreas de interesse sob o domínio de programação em shell que não foram tocadas por este documento, mas que poderiam trazer benefícios de aprendizado e valor agregado incluem controle de processos, navegação, bem como customização do ambiente de desenvolvimento.

Apêndice A

Scripts para Análise de Inventário

Os scripts a seguir foram implementados a fim de se fazer a análise de inventário do patrimônio da universidade. Estes scripts também estão à disposição em `https://gitlab.com/israel.prates/ci320-scripts`.

A.1 Obtenção de Dados de Entrada

Este script consiste apenas da chamada ao comando scp (secure copy), que, por sua vez, permite a cópia de arquivos remotos via protocolo SSH. O script abaixo foi primariamente utilizado para encapsular o cascateamento das atualizações do arquivo de entrada, visto que múltiplas modificações foram aplicadas sobre o mesmo em seu ciclo de vida.

```
1 #!/bin/bash
2
3 scp 'ifp15@ssh.inf.ufpr.br:~marcos/tmp/patrimonio.csv' './file-01.csv'
```

Script A.1: inventory-fetch-source.sh

A.2 Análise de Inventário

Abaixo o script que automatiza o processo de filtragem e separação de itens de inventário por localidade. Os principais conceitos aplicados foram uso de funções, tratamento de sinais, uso de arquivos temporários, expansão de variáveis e expressões regulares.

```
1 #!/bin/bash
2
3 ##############################################################################
4 #                                Declarations                                #
5 ##############################################################################
6
```

```
# Read-only variable with a regex that matches any lines with one or more pairs
# of double quotes followed by a semicolon.
declare -r pattern='^(("[^";]*")+;)+$'
# Source name.
declare -r source='./file-01.csv'
# Temporary file. Semicolons removed from field contents.
declare -r filtered="$(mktemp filtered-temp-XXXXXX)"
# Temporary file. Locations (5th column).
declare -r locations="$(mktemp locations-temp-XXXXXX)"
# Output directory.
declare -r output_dir='./output/'

################################################################################
#                              Function definitions                            #
################################################################################

# Function to print an error message to stderr.
function print_error() {
    printf "%s\n" "$*" >&2
}

# Clean-up function. Silently delete any file(s) passed as argument.
function clean() {
    rm -f "$@"
}

################################################################################
#                                Start of script                               #
################################################################################

# Define the clean-up function as the handler to the exit event.
trap 'clean ${filtered} ${locations}' EXIT

# Check whether the output directory exists. Abort if doesn't.
if [ ! -d "${output_dir}" ] || [ ! -f "${source}" ]; then
    print_error "${0}: Error: Could not locate necessary files. Aborting."
    exit 1
fi

grep -E "${pattern}" "${source}" > "${filtered}"

cut -d';' -f5 "${filtered}" | sort -u > "${locations}"

# Run on a subshell in order to preserve IFS (internal field separator).
(
    IFS=''
    while read -r line || [[ -n "${line}" ]]; do
        grep -E "${line};$" "${source}" | sort > "${output_dir}${line//\"/}.csv"
    done < "${locations}"
)

# EOF
```

Script A.2: inventory.sh

Apêndice B

Análise de Volume de Matrículas

Os scripts a seguir foram implementados a fim de se fazer a análise da progressão do volume de matrículas por semestre. Estes scripts também estão à disposição em `https://gitlab.com/israel.prates/ci320-scripts`.

B.1 Obtenção de Dados de Entrada

O script abaixo, além de automatizar o processo de cascateamento de modificações dos arquivos de entrada, também faz modificações à estrutura de pastas e renomeia os arquivos relevantes para o problema. A principal motivação ao fazê-lo foi melhorar a legibilidade final do código produzido ao padronizar o idioma de escrita.

```
1  #!/bin/bash
2
3  ################################################################################
4  #                                 Declarations                                 #
5  ################################################################################
6
7  # Path to the source file. Compacted with tar and gzip.
8  declare -r source='/home/c3sl/marcos/tmp/dadosmatricula.tgz'
9  # Path to the destination file. Compacted with tar and gzip.
10 declare -r destination='/nobackup/bcc/ifp15/registration-data.tgz'
11 # Extracted directory path.
12 declare -r extracted='/nobackup/bcc/ifp15/registration-data/'
13 # Symbolic link to the extracted directory.
14 declare -r link_name='./registration-data'
15
16 ################################################################################
17 #                                Start of script                               #
18 ################################################################################
19
20 # Check whether the source file exists and is readable.
21 # If not, then print an error message and exit with failure.
22 [ -f "${source}" ] || {
23     echo "${0}: Error: unable to locate source file." >&2
```

```
    exit 1
}

# Check whether the extracted directory exists. If it doesn't, then create it.
[ -d "${extracted}" ] || mkdir "${extracted}"

# Create a symbolic link to the extracted directory.
ln -fsT "${extracted}" "${link_name}"

# Copy the source file to the destination path.
cp "${source}" "${destination}"

# Extract the source file to the extracted directory while stripping its root.
tar xf "${destination}" -C "${extracted}" --strip-components 1

# Rename every file whose extension is .dados so it reads .dat instead.
find "${extracted}" -name '*.dados' -exec rename -f 's/\.dados/\.dat/i' {} \;
# EOF
```

Script B.1: registration-fetch-source.sh

B.2 Análise Simplificada

O script que segue não reporta a variação no volume de matrículas agrupado por disciplina, mas apenas o total por semestre. Os principais conceitos empregados foram uso de funções, tratamento de sinais, uso de arquivos temporários, expansão de variáveis, expressões regulares e arranjos associativos.

```
#!/bin/bash

################################################################################
#                                Declarations                                  #
################################################################################

# Path to the data directory.
declare -r data_directory='./registration_data/'
# Pattern that describes the data files' extension.
declare -r extension_pattern='*.dat'
# Path to the output file.
declare -r output='./output.dat'
# Dictionary holding the paths to the semester temporary files.
# Indexed by semester.
declare -A temp_semesters=()

################################################################################
#                            Function definitions                              #
################################################################################

# Function to print an error message to stderr.
function print_error() {
    printf "%s\n" "$*" >&2
}

# Clean-up function. Silently delete any file(s) passed as argument.
function clean() {
    rm -f "$@"
}

################################################################################
#                              Start of script                                 #
```

```
33 ################################################################################
34
35 # Check whether the data directory exists. If it does not, exit with failure.
36 if [ ! -d "${data_directory}" ]; then
37     print_error "${0}: Error: could not find '${data_directory}' directory."
38     exit 1
39 fi
40
41 # Trap the exit signal and hook the clean function to it.
42 # Its arguments are the contents of the temp_semesters dictionary, expanded only
43 # when the signal is received (note the quotes).
44 trap 'clean ${temp_semesters[@]}' EXIT
45
46 # Iterate through the files under the data directory whose extensions match the
47 # extension_pattern.
48 for file in $(find "${data_directory}" -name "${extension_pattern}"); do
49     # Remove, starting from the front, anything until the last slash, inclusive.
50     semester="${file##*/}"
51     # Remove, starting from the back, anything until the last dot, inclusive.
52     semester="${semester%%.*}"
53     temp="${semester}.tmp"
54     # Use the semester as key in the dictionary to store the temp file path.
55     temp_semesters["${semester}"]="${temp}"
56     # Exclude the first line by indicating to head that the second line from
57     # the file should be used as starting point.
58     # Get the second column from the piped output and append it to the file
59     # specified by temp. The second column is the GRR. The separator is the
60     # colon character (:).
61     tail --lines=+2 "${file}" | cut -d':' -f2 >> "${temp}"
62 done
63
64 # Truncate (empty) the output file.
65 > "${output}"
66
67 # Iterate through the  keys of the temp_semesters dictionary.
68 for semester in "${!temp_semesters[@]}"; do
69     # Echo the semester and the amount of unique registrations in it. Append
70     # the result to the output file.
71     echo "${semester}:$(sort -u ${temp_semesters[${semester}]} | wc -l)"
72         >> "${output}"
73 done
74
75 # Sort (in place) the output lines.
76 sort -n --output="${output}" "${output}"
77 # EOF
```

Script B.2: registration-simple.sh

B.3 Análise Detalhada

Este script passa a distinguir disciplinas, de modo a agregar detalhe aos dados reportados. Os principais conceitos associados são tratamento de opções da linha de comando, navegação na árvore de diretórios e encadeamento condicional de comandos.

```
1 #!/bin/bash
2
3 ################################################################################
4 #                                Declarations                                  #
5 ################################################################################
```

```
6
7  declare -r root='./registration_data/'
8
9  # Underlying flag for options.
10 declare header=false
11 declare pretty_print=false
12 declare options=()
13
14 # Command line options parsing.
15 options=$(getopt --options hp --long header,pretty-print --name "${0}" -- "$@")
16 # If last command failed, then exit script with failure.
17 [ "$?" != 0 ] && exit 1
18
19 # Set underlying variables.
20 while true; do
21     case "${1}" in
22         -h | --header ) header=true; shift ;; # Header option.
23         -p | --pretty-print ) pretty_print=true; shift ;; # Pretty print option.
24         -- ) shift; break ;; # Last option has been read. Break.
25         * ) break ;; # Anything else. Break.
26     esac
27 done
28
29 ################################################################################
30 #                              Function definitions                            #
31 ################################################################################
32
33 # Function to print an error message to stderr.
34 function print_error() {
35     printf "%s\n" "$*" >&2
36 }
37
38 # Function to perform a specified function on a stream.
39 function map_function() {
40     local -r mapped_function="${1}"
41     while read line; do
42         # Call the function with the line as its argument.
43         "${mapped_function}" "${line}"
44     done
45 }
46
47 # Function to subtract one from its argument and print the result.
48 function minus_one() {
49     local -ri value="${1}"
50     printf "%d\n" "$((${value} - 1))"
51 }
52
53 # Count registrations for all disciplines.
54 function count_registrations() {
55     local -r dir="${1}"
56     pushd "${dir}" > /dev/null # Go down into dir.
57     printf "%s" "$(grep -c '.*' * | # Count matches on each file.
58         cut -d':' -f 2 | # Get the second column, which is the count itself.
59         map_function minus_one | # Subtract one from each line.
60         tr '\n' ' ' | # Translate line feeds to spaces.
61         sed 's/ $//')" # Remove trailing space.
62     popd > /dev/null # Go up into dir structure.
63 }
64
65 # Function to create the header, where each column is a semester.
66 function compose_header() {
67     local dir="${1}"
68     local ext='*.dat'
69     for item in $(find "${dir}"/* -name "${ext}"); do
```

```
        item="${item##*/}"
        printf "%s\n" "${item%%.*}"
    done | sort -u | tr '\n' ' ' | sed 's/ $//'
}

################################################################################
#                                Start of script                               #
################################################################################

# Check whether the root directory exists. If not, then exit with failure..
if [ ! -d "${root}" ]; then
    print_error "${0}: Error: could not find '${root}' directory."
    exit 1
fi

# Subshell. Used for redirection ahead in the script.
(
    # Print the header if the corresponding option has been provided.
    if [[ "${header}" = true ]]; then
        printf "Discipline/Semester %s\n" "$(compose_header ${root})"
    fi
    pushd "${root}" > /dev/null # Go into root directory.
    # Count the registrations on each discipline.
    # Each directory is the name of a discipline.
    for dir in $(ls .); do
        printf "${dir} %s\n" "$(count_registrations ${dir})"
    done
    popd > /dev/null
) | (
        # Check whether the pretty_print option has been checked. If true, then
        # use column to format the output, otherwise simply display the result.
        [[ "${pretty_print}" = true ]] && column -t || cat
    )
# EOF
```

Script B.3: registration-detailed.sh

Apêndice C

Sumarização de Registro de Firewall

Os scripts a seguir foram implementados a fim de se automatizar o processo de sumarização de arquivos de registro de firewall de maneira eficiente. Estes scripts também estão à disposição em `https://gitlab.com/israel.prates/ci320-scripts`.

C.1 Filtragem do Arquivo de Entrada

O script em AWK abaixo tem por finalidade computar e reportar a quantidade de ocorrências de regras filtradas no firewall da universidade. Caso um número excepcional de ocorrências seja observado, uma mensagem deve ser enviada ao administrador notificando a ocorrência. Dado que todo o processo de filtragem deve ocorrer rapidamente – mesmo para arquivos grandes – a linguagem AWK foi adotada para esta implementação de filtro, visto que é uma linguagem própria para processamento de texto.

Conceitos relevantes empregados na implementação deste script são uso de variáveis, arranjos, controle de fluxo, laços e chamadas de sistema em AWK.

```
1  #!/usr/bin/awk -f
2
3  # Print s in red.
4  # Messes sorting, since the escape sequence is part of the string.
5  function red(s) {
6      printf "\033[1;31m" s "\033[0m"
7  }
8  # Print s in green.
9  # Messes sorting, since the escape sequence is part of the string.
10 function green(s) {
11     printf "\033[1;32m" s "\033[0m"
12 }
13 BEGIN {
14     types_list="./firewall-types.txt"
15     send_mail_script="./firewall-send-mail.sh"
16     threshold=20000
```

```
17
18      # Initialize the array.
19      while ((getline type < types_list) > 0) {
20          occurrences[type]=0
21      }
22  }
23  {
24      # Increment the count of occurrences for a given type.
25      occurrences[$6]++
26  }
27  END {
28      # Loop through the keys.
29      for (type in occurrences) {
30          # Check for abnormal amount of occurrences
31          if (occurrences[type] >= threshold) {
32              # Send the notification messages.
33              # Setting either of below variables will prevent the messages from
34              # being sent.
35              if (!suppress_email && !nomsg) {
36                  system(send_mail_script " " type " " occurrences[type])
37              }
38              # Report occurrences exceeding the threshold in red.
39              print red(type " " occurrences[type])
40          }
41          else {
42              # Report normal occurrences in green.
43              print green(type " " occurrences[type])
44          }
45      }
46  }
```

Script C.1: firewall.awk

C.2 Envio de Mensagens de Alerta

Este script envia um email de notificação de acordo com um modelo. O tipo e a quantidade de ocorrências que se deseja notificar devem ser informados durante a chamada deste script.

```
1  #!/bin/bash
2
3  declare -r type="${1}"
4  declare -r occurrences="${2}"
5
6  # The file delimiter could be anything, not only EOF.
7  mail -s 'Abnormal Number of Blocked Access Attempts' 'ifp15@inf.ufpr.br' <<EOF
8  This message is to notify you that an abnormal number of requests has been
       blocked on your firewall. Type and number of occurrences below.
9
10 ${type} ${occurrences}
11 EOF
```

Script C.2: firewall-send-mail.sh