Capítulo 2. Bash *shellscripting*

Frederico Schmitt Kremer [1], Luciano da Silva Pinto [1]

[1] *Laboratório de Bioinformática e Proteômica, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas*

Objetivos: Introduzir a programação de *scripts* de *shell* (*shellscripts*) usando o interpretador de comandos Bash.

O que é um *script*?

Como dito no capítulo anterior, alguns programas precisam ser compilados para serem executados, sendo este processo necessário para que o **código fonte** (entendível por humanos) seja convertido em um conjunto de **instruções binárias** (entendível por máquinas). O processo de compilação gera um programa que é executável em um ambiente (*hardware* + sistema operacional) específico, não sendo este **portável** para outras arquiteturas (ex: Windows 32 bits para Windows 64 bits) ou sistemas operacionais (ex: Linux para Mac OS). Existem diferentes linguagens de programação que precisam ser compiladas para que seus códigos se tornem programas executáveis, sendo C, C++ e Fortran algumas das mais populares.

Scripts, por outro lado, são programas de computador escritos na forma de um **arquivo de texto** e executados por meio de um **interpretador**. Um *scripts* não precisa ser compilado, mas o seu interpretador sim. Isso significa que, tem teoria, um *script* escrito em uma máquina pode ser executado em qualquer sistema ou *hardware* deste que para este exista um interpretador para a linguagem de interesse. Além dos *shellscripts*, muitas outras linguagens de programação também são utilizadas na forma de código interpretado, como as linguagens Python (amplamente utilizada para aplicações comerciais e acadêmicas), Perl (usada para processamento de texto e web) e R (usada análises estatísticas e mineração de dados). O próprio Javascript, ubiquo nos na Web e que permite que as páginas de internet sejam interativas, é uma linguagem interpretada, o que permite que um mesmo site seja visualizado independente das configurações do sistema operacional.

Editores de Texto

Para a escrita de um *script* é necessário apenas um editor de texto, que pode ser uma ferramenta simples, como o **bloco de notas** do Microsoft Windows e **Gedit** do Linux, ou expecializadas para desenvolvimento de programas, como os editores **Sublime Text**, **Atom**, **Vi** / **Vim**, **Pico** / **Nano**, **EMACS** e **Notepad++**.

"Ola, Mundo"

Em 1978, Brian Kernighan e Dennis Ritchie, ambos funcionários do Bell Lab da AT&T, publicaram o livro "*The C Programming Language*", que viria a se tornar a referência *de facto* para a linguagem de programação C, desenvolvida por Ritchie. De forma a apresentação o princípios mais fundamentais da linguagem, Kernighan apresenta um pequeno programa que quando executado mostra na tela a mensagem "hello, world". O código em C deste programa é:

```
#include <stdio.h>

main( )
{
      printf("hello, world\n");
}
```

Neste caso, a função "printf" indica que o texto escrito dentro dos parênteses deve ser mostrado na tela, e o "\n" no final da frase é uma indicação de que após isso deve ser feita uma quebra de linha.

Deste a publicação deste livro, o programa "hello, world" vem sendo usado como um dos principais exemplos introdutórios de diferentes linguagens de programação. No caso de um *script* de *shell*, por exemplo, esta mesma tarefa poderia ser feita com apenas uma linha:

```
echo "hello, world"
```

Para executar este programa, basta escrever-lo em um arquivo de texto (ex: "`programa.sh`"), e depois utilizar o comando `bash [nome do programa]`.

```
$ bash programa.sh
hello, world
```

Apesar de ser um exemplo simples, o fato de ser necessário um número menor de linhas para executar esta tarefa em *shellscript*, quando comparado com o código equivalente em C, demonstra uma das principais características deste tipo de linguagem: **concisão**. Por conta disso, a escrita de um *script* exige normalmente pouco tempo, e já que o código não precisa ser compilado, pode ser facilmente modificado, corrigido e otimizado. Outra característica importante destas linguagens é a flexibilidade,

Executando comandos

O nome Bash diz respeito tanto à linguagem de scripts quando ao programa que interpreta os comandos escritos nesta linguagem, seja através de um sistema interativo

(ex: terminal) ou de um conjuntos de comandos pré-estabelecidos (ex: *script*). Desta forma, por se tratar do mesmo ambiente, os comandos executáveis através de um script são os mesmo que seriam executáveis pela interface de linha de comandos. Virtualmente, todos os comandos e operações realizáveis no Linux podem ser também feitas através de *shellscripts*, o que torna estes ferramentas poderosas para a automatização de tarefas e administração de sistemas.

Um *shellscript* pode ser escrito em qualq.uer **editor de texto**, e sua execução depende apenas do seu interpretador. Desta forma, estes são facilmente portáveis para diferentes máquinas, sistemas e versões do Linux. De fato, salvo uma ou outra diferença, até mesmo diferentes interpretadores do shell (ex: Bash, C shell, Korn shell) tem uma certa compatibilidade, visto que muitos dos comandos presentes são meras incorporações e/ou extensões de funcionalidades do próprio sistema.

Executando comandos

execução dos comandos presentes em um *shellscript* é feita serialmente, ou linha-por-linha. Para entender isso, vamos fazer um pequeno *script* chamado "concatenar.sh", que realiza 3 operações: (1) cria um diretório chamando "resultados"; (2) faz a concatenação de dois arquivos, "reads_1.fastq" e "reads_2.fastq", e gera um novo arquivo, "reads_merged.fastq"; (3) move o arquivo gerado para a pasta criada. Neste caso, o conteúdo do *script* seria:

```
mkdir resultados
cat reads_1.fastq reads_2.fastq > reads_merged.fastq
mv reads_merged.fastq resultados/
```

Note que, já que a transferência do arquivo gerado depende do resultado dos comandos `mkdir` e `cat`, então é importante que uma nova linha seja executada apenas quando a anterior já tenha sido finalizada. Este tipo de execução é chamada **sincrona**, e é adotada como padrão para praticamente qualquer linguagem de programação (apesar de em alguns casos ser interessantes executar novas tarefas enquanto outra ainda está pendente).

Comentários

Comentários são trechos não-executáveis adicionados ao código de um softwares, normalmente com a finalidade de descrever a função daquilo que está sendo implementado. Em *shellscript*, assim como em outras linguagens de script como Python e Perl, os comentários podem ser definidos através do símbolo "#". Quando adicionado a uma linha, tudo aquilo que está escrito após este símbolo será ignorado pelo interpretador.

Considerando o *script*:

```
# echo "isto é um comentário"
echo "isto não é um comentário"
```

O resultado da sua execução será apenas a frase "isto não é um comentário".

Shebang ("#!")

O *shebang* é um comentário especial que é adicionado na primeira linha do *script* para indicar qual interpretador deve ser utilizado para a sua execução. Apesar de ser possível chamar o interpretador explicitamente (ex: "`bash meu_script.sh`"), em algumas situações pode ser útil se fazer isso de forma implícita (ex: "`./meu_script.sh`").

Como scripts são programas de computador, é possível informar ao sistema que estes não são apenas arquivos de texto, mas também arquivos executáveis, assim como aqueles que são gerados após a compilação de um código-fonte. Para fazer isso, basta se utilizar o comando `chmod +x [caminho para o script]`, como por exemplo:

```
$ chmod +x meu_script.sh
```

Agora, a execução do script pode ser feita através da seguinte forma:

```
$ ./meu_script.sh
```

O *shebang* é utilizado por diversas linguagens de programação interpretadas que rodam em sistemas Linux, como Python, Perl e R, além é claro das linguagens de *shellscript*. No caso do Bash, uma das *shebangs* mais comuns é a "`#!/usr/bin/bash`", mas nem sempre o interpretador do Bash está instalado nesta pasta, o que pode acabar resultamdo em um erro de execução do *script*. Para evitar este tipo de problema, é comum utilizar o caminho para um outro programa, `env`, que guarda a localização de cada programa que está disponível na lista de diretórios de programas mapeados pelo sistema (chamada `$PATH`). Para utilizar o `env`, basta adicionar ele no lugar to nome do interpretador, e colocar o nome deste logo depois, separando-os por um espaço. Desta forma, a *shebang* que seria "`#!/usr/bin/bash`" passa a ser "`#!/usr/bin/env bash`". Por ser uma ferramenta básico do sistema, a localização do `env` é muito menos variável de distribuição para distribuição se comparado aos interpretadores de comando.

Variáveis

Em programação, **váriáveis** representação espaços na memória do computador que são utilizados para armazenar temporariamente determinados valores, sendo estes sujeitos a alterações futuras. Usualmente, é comum se classificar as variáveis em diferentes

categorias, como caracteres (*char*), textos (*string*), números inteiros (integers), números reais (*float*), valores booleanos (*boolean*). De acordo com a linguagem de programação, o tipo de uma variável pode ser estático (**tipagem forte**) ou variável (**tipagem fraca**). No caso de linguagens com tipagem forte, cada variável possui um tipo definido que não pode ser alterado durante a execução do programa. A tipagem forte é comum em liguagens compiladas, uma vez que o programa precisa alocar espaço na memória física do computador para executar suas funções. Desta forma, o nome o o tipo de uma variável precisam ser **declarados** explicitamente antes de sua utilização. No caso da tipagem fraca, por outro lado, o tipo de uma variável é mutável, e em alguns casos não é nem necessário declarar explicatamente seu nome antes da utilização. A tipagem fraca é bastante comum em linguagens interpretadas, uma vez que permite uma maior flexibilidade e agilidade durante a escrita do código. Para criar uma variável em um *shellscript*, basta se escrever o seu nome, seguido de um sinal de igual ("=") (sem espaço), e por fim o valor que deseja atribuir a esta.

```
#!/usr/bin/env bash
# exemplo de uma variável
nome="frederico"
```

O valor associado armazenado em uma variável pode ser utilizado como parte de um comando que será executado pelo *script*, sendo para isso necessário se indicar o nome da variável de interesse precedido por um símbolo de dólar ("$") (sem espaço). Como exemplo:

```
#!/usr/bin/env bash
# exemplo de uma variável
nome="Jaspion"
echo $nome
```

O resultado destes comandos seria a mensagem "Jaspion" na tela. Caso o símbolo de dólar não fosse adicionado, a mensagem mostraria seria "nome", visto que o interpretador entenderia o parâmetro passado como um **valor literal**, e não como uma variável.

Como o nome diz, variáveis não tem um valor estático, ou seja, seu valor pode ser atualizado ao longo do **tempo de execução** do *script*. Para entender melhor isso, preste atenção no exemplo abaixo:

```
#!/usr/bin/env bash
# exemplo de uma variável
nome="Jaspion"                # inicializa a variável
echo $nome                    # mostra a mensagem "Jaspion"
nome="Jiraya"                 # atualiza a variável
echo $nome                    # mostra a mensagem "Jiraya"
```

Desvio condicional: *if … then … (else …) fi*.

Para uma linguagem de programação ser considerada "completa", ou melhor, **Turing completa**, existem alguns pré-requisitos, sendo um deles relacionados a capacidade de se executar uma determinada tarefa como consequência de uma afirmação ser verdadeira ou falsa. Este tipo de escolha é também denominada "desvio condicional", pois parte das funções que serão executadas pelo programa dependerão de uma determianda condição. Em Bash, um desvio condicional simples pode ser feito através da seguinte estrutura:

```
if [ CONDIÇÃO ]; then
      # executar estas tarefas caso a condição seja verdadeira
fi
```

A **condição** pode ser uma comparação entre dois valores para ver se estes são iguais ou diferentes, ou para se verificar se um determinado arquivo ou diretório existe, por exemplo. Alguns exemplos de operações de comparação estão listados na Tabela 1. Alguns exemplos de outras funções uteis disponíveis no *if* do Bash estão apresentadas na Tabela 2.

Tabela 1. Operações de comparação entre variáveis disponíveis no Bash.

Operação	Significado	O que compara	Exemplo
-eq	"igual"	Números	`if [ "$a" -eq "$b" ] ...`
-ne	"não é igual"	Números	`if [ "$a" -ne "$b" ] ...`
-gt	"maior que"	Números	`if [ "$a" -gt "$b" ] ...`
-ge	"maior ou igual"	Números	`if [ "$a" -ge "$b" ] ...`
-lt	"menor que"	Números	`if [ "$a" -lt "$b" ] ...`
-le	"menor ou igual"	Números	`if [ "$a" -le "$b" ] ...`
<	"menor que"	Números	`if (( "$a" < "$b" )) ...`
<=	"menor ou igual"	Números	`if (( "$a" <= "$b" )) ...`
>	"maior que"	Números	`if (( "$a" > "$b" )) ...`
>=	"maior ou igual"	Números	`if (( "$a" => "$b" )) ...`
=, ==	"igual"	Texto	`if [ $nome == "Jaspion" ] ...`
=, ==	Busca de padrões	Texto	`if [ $nome == J* ] ...`
!=	"Diferente"	Texto	`if [ $nome != "Jaspion" ] ...`
>	"maior que" (ordem alfabética)	Texto	`if [[ $nome > "Jaspion" ]] ...`
<	"menor que" (ordem alfabética)	Texto	`if [[ $nome < "Jaspion" ]] ...`

Tabela 2. Exemplos de operações disponíveis no Bash.

Operação	Significado	O que compara	Exemplo
-d	"o diretório existe"	Texto	`if [ -d $diretorio ] ...`
-f	"o arquivo existe"	Texto	`if [ -f $arquivo ] ...`
-z	"valor (da variável) é nulo"	Qualquer valor	`if [ -z $valor ] ...`
-n	"valor (da variável) não é nulo"	Qualquer valor	`if [ -n $valor] ...`

Como dito, o *if* avalia se uma declaração é verdadeira ou falsa, e, se e somente se esta for verdadeira, os comandos presentes neste bloco serão executados. Entretanto, é possível se inverter a lógica e testar se uma determinada afirmação é falsa através do uso de um ponto de exclamação ("!") antes da operação. Tomando como exemplo a opção "-d" do if, apresentada na Tabela 2, podemos utilizar esta estrutura para avaliar se um determinado direfortório não existe e, caso não exista, criar ele através do comando `mkdir`. Desta forma, nosso *script* será:

```
#!/usr/bin/env bash
diretorio=genomas
if [ ! -d $diretorio ]; then
     echo "criando diretorio $diretorio"
     mkdir $diretorio
fi
echo "o diretorio esta criado"
```

A estrutura *if* executa um bloco de comandos caso uma determinada condição seja verdadeira, mas muitas vezes pode ser necessário se executar um outro conjunto de comandos apenas nos casos em que esta condição for falsa. Desta forma, o programa oferece dois "caminhos" possíveis, e só um será executado, sendo a "escolha" feita de acordo com os resultados da avaliação da mesma expressão. Para criar este caminho alterantivo é utilizada a estrutura *else* junto ao bloco *if*, sendo a sua estrutura definida por:

```
if [ CONDIÇÃO ]; then
     # executar estas tarefas caso a condição seja verdadeira
else
     # executar estas tarefas caso a condição seja falsa
fi
```

Mantendo o exemplo do *script*

```
#!/usr/bin/env bash
diretorio=genomas
if [ ! -d $diretorio ]; then
     echo "criando diretorio $diretorio"
     mkdir $diretorio
else
     echo "O diretorio $diretorio ja existe"
fi
```

Estrutura de repetição: *for … do … done*.

Outros elementos importantes de uma linguagem de programação são as chamadas estruturas de repetição, ou *loop* ("laço"), que pemitem que um um determinado conjunto

de comandos seja repetido até que uma uma certa condição determine a sua interrupção. Uma das formas mais simples de se fazer isso no Bash é através da estrutura *for,* que permite se iterar sobre uma lista de valores.

```
for NOME_DA_VARIAVEL in LISTA_DE_VALORES; done
      # executar as tarefas
done
```

Um exemplos do uso do *for* para se executar o mesmo comando para dois valores diferentes está indicado logo abaixo. Neste caso, a cada volta do *loop* um novo valor é atribuido à variável "arquivo".

```
#!/usr/bin/env bash
# script: exemplo_for_1.sh
for arquivo in arquivo_1.fastq arquivo_2.fastq; do
      echo $arquivo
done
```

Desta forma, a execução deste *script* resultaria em:

```
$ ./exemplo_for_1.sh
arquivo_1.fastq
arquivo_2.fastq
```

Outro praticidade do for no Bash é a possibilitade de se iterar sobre os arquivos e sub-diretórios presentes em uma determidada pasta. Para se fazer isso, tome como exemplo o seguinte *script*:

```
#!/usr/bin/env bash
# script: exemplo_for_2.sh
for arquivo in arquivos_sequenciamento/*;
      echo $arquivo
done
```

Caso o diretório "arquivos_sequenciamento" tenha três arquivos ("cepa_1.fastq", "cepa_2.fastq", "cepa_3.fastq"), a execução deste *script* resultaria em:

```
$ ./exemplo_for_2.sh
arquivos_sequenciamento/cepa_1.fastq
arquivos_sequenciamento/cepa_2.fastq
arquivos_sequenciamento/cepa_3.fastq
```

Argumentos

Argumentos são parâmetros passados a um determinado programa durante a sua chamada, ou seja, como parte da linha de comando que o executa. Um exemplo da utilização de argumentos é:

```
$ ./montar_genoma.sh leituras_sequenciamento.fastq
echo "arquivo de entrada: $1"
```

Neste caso, o *script* "montar_genoma.sh" recebe como parâmetro o nome de um arquivo contendo leituras de sequenciamento de um genoma a ser montado. Internamento, todos os argumentos passados durante a chamada de um programa são transformados em variáveis, que podem ser acessadas de forma similar as variáveis criadas pelo usuário, através do sinal de dólar "$" seguido do **índice do argumento** de interesse (sem espaço). Os argumentos são analisados na ordem que foram escritos, sendo o primeiro denominado "$1", o segundo "$2", e assim por diante. Desta forma, para o script "montar_genoma.sh" acessar o nome do arquivo passado pelo usuário, por exemplo, o programa precisa pegar os valor armazenado na variável "$1".

```
#!/usr/bin/env bash

#mostra na tela o nome do arquivo de dados de sequenciamento
echo "arquivo de entrada: $1"
```

Os argumentos passados para um script podem também ser acessados iterativamente em um bloco *for* através da variável "$@". Desta forma, para se imprimir cada um dos argumentos passados para um *script* (ex: "iterar_argumentos.sh") basta se utilizar a seguinte sintaxe:

```
#!/usr/bin/env bash
for argumento in $@; do
      echo $argumento
done
```

Um exemplo da execução deste *script* é:

```
$ ./ iterar_argumentos.sh argumento_1 argumento_2 argumento_3
argumento_1
argumento_2
argumento_3
```

Recebendo resultados de programas

Muitos programas, inclusive funções nativas do próprio Linux, imprimem na tela informações úteis que podem ser acessadas através de *scripts* e utilizadas como parte de determinados comandos. Para se acessar estes valores é necessário se utilizar a sintaxe "$(comando)", e estes podem ser transferidos para uma variável através da sintaxe "variavel=$(comando)". Um exemplo da utilização desta funcionalidade está ilustrada no *script* abaixo ("root.sh"), que identifica se o mesmo está sendo executado como `root` ou não.

```
#!/usr/bin/env bash
#script: root.sh
user=$(whoami)
if [ $user == "root" ]; then
     echo "você é o root"
else
     echo "você não é o root"
fi
```

Desta forma, executando o programa com o sudo e sem o sudo resultarão em mensagens diferentes.

```
$ ./root.sh
você não é o root
$ sudo ./root.sh
você é o root
```

Outros exemplos de funções úteis do Linux que podem ser incorporadas em um *scripts* através desta mesma abordagens são: `pwd` (atual diretório de trabalho) (abordado no **capítulo 1**), `basename` (apenas o nome de um arquivo, sem o caminho de diretórios), `dirname` (caminho para o diretório de um determinado arquivo) e readlink –f (caminho absoluto para um arquivo ou diretório). Para entender o funciomanto dos comandos `basename`, `dirname` e `readlink -f` vamos ver alguns exemplos:

Considere uma pasta chamada "minha_pasta" localizada no diretório "/home/bioinfo", uque contêm um arquivo de texto "arquivo.txt".

```
$ ls minha_pasta/
arquivo.txt
$ basename minha_pasta/arquivo.txt
arquivo.txt
$ dirname minha_pasta/arquivo.txt
minha_pasta
$ readlink -f minha_pasta/arquivo.txt
/home/bioinfo/minha_pasta
```

Um exemplo de *script* (“funcoes_linux.sh”) que utiliza destas funções para listar e manipular o nome dos arquivos presentes eu sua pasta está apresentado abaixo.

```
#!/usr/bin/env bash
script: funcoes_linux.sh

# - obter o caminho para o atual diretório de trabalho
# - gravar na variável “meu_diretorio”

meu_diretorio=$(pwd)

# - iterar sobre os arquivos presentes na pasta

for arquivo in meu_diretorio/*; do

     # - mostrar o caminho completo do arquivo

     echo $arquivo

     # - mostrar apenas o nome do arquivo

     echo $(basename $arquivo)

     # - mostrar apenas o nome do diretório do arquivo

     echo $(dirname $arquivo)

     # - mostrar o caminho absoluto do arquivo

     echo $(readlink –f $arquivo)

done
```