Casamento de padrões no shell do GNU/Linux

Blau Araujo

Índice

1 - Padrões

Padrões são atributos que expressam algum tipo de regularidade, algo em comum que pode ser encontrado em elementos distintos de um mesmo tipo. Observe a imagem abaixo:

Nela, vemos 5 bolas de futebol diferentes, mas algumas delas compartilham um atributo em comum: consegue identificar que atributo é esse? Se você respondeu as cores dos pentágonos do centro, você acabou de identificar padrões!

Agora, imagine que as bolas estão na prateleira de uma loja e o vendedor pergunta qual delas você quer comprar. Digamos que você responda: *a que tem um pentágono vermelho no centro*. Então, de todas as bolas expostas, o vendedor leva até você as duas que possuem um pentágono vermelho e pergunta: *qual delas*?

Sim, porque, apesar das duas bolas atenderem ao padrão que você **descreveu**,uma tem os demais pentágonos pretos e a outra azuis. Quando falamos de padrões de texto, não é diferente.

1.1 - Padrões de texto

Textos são compostos de caracteres alfabéticos maiúsculos e minúsculos, dígitos numéricos, espaços em branco e outros símbolos gráficos, mas todos esses caracteres são limitados e, eventualmente, irão repetir-se conforme algum padrão.

Por exemplo, observe com atenção as palavras abaixo:

```
carro abacate alicate cerca cabelo porca
```

Todas são escritas com caracteres alfabéticos minúsculos, o que já é um padrão em comum, mas é um padrão insuficiente se quisermos diferenciá-las de outras palavras que, porventura, estejam no mesmo texto.

Observe com mais atenção ainda: o que essas palavras têm em comum?

Coincidentemente, em todas elas existe a ocorrência do caractere `c` seguido do caractere `a`:

```
[ca]rro aba[ca]te ali[ca]te cer[ca] [ca]belo por[ca]
```

Sendo assim, se tivéssemos que diferenciar estas palavras de todas as outras em um contexto maior, um bom começo seria dizer: *eu quero as palavras que contenham a sequência de caracteres* `c-a`. Por outro lado, isso ainda poderia ser o mesmo que escolher as bolas que têm pentágonos vermelhos: a **descrição** daquilo que queremos pode ser insuficiente. Por isso, quanto mais específica a nossa busca, mais atributos precisarão ser descritos.

Vamos tentar agrupar as palavras de outras formas, segundo outros atributos, por exemplo, de acordo com a posição em que a sequência `c-a` aparece nas palavras:

```
[ca]rro   [ca]belo   → [ca] aparece no começo
cer[ca]   por[ca]    → [ca] aparece no final
aba[ca]te ali[ca]te  → [ca] aparece no meio
```

Mas essas ainda não são as únicas descrições possíveis!

Repare:

```
por[ca]
cer[ca]
aba[ca]te
ali[ca]te
```

Em todas elas, a sequência `c-a` é precedida por três outros caracteres. Alias,

`alicate` e `abacate` têm outras coisas em comum:

- Possuem sete caracteres;
- Começam com o caractere `a`;
- Terminam com a sequência `c-a-t-e`.

Como você pode ver, nós podemos utilizar uma infinidade de atributos para descrever as sequências de caracteres que nos interessam em um texto e, quanto mais específica a nossa descrição, menos perguntas o vendedor da loja de bolas de futebol fará, economizando etapas no processamento da compra.

Trazendo essa analogia para a computação, quanto mais específicos na descrição de um padrão, menos processos intermediários serão necessários para filtrarmos aquilo que nos interessa em um texto.

1.2 - Casamento de padrões

Existe uma expressão em inglês que está virando moda entre desenvolvedores brasileiros: *pattern matching*. Apesar da aura mística que o jargão invoca, trata-se do nosso bom e velho ***casamento de padrões***, que diz respeito às técnicas utilizadas para comparar descrições de padrões com sequências de caracteres. Isso porque, quando a descrição de um padrão coincide com alguma sequência de caracteres, nós dizemos que houve um *casamento* (em inglês, *matching*).

Pensando pelo lado das técnicas, os casamentos de padrões sempre irão envolver três componentes:

- Uma fonte de dados em texto;
- Uma linguagem para a descrição do padrão;
- E uma ferramenta que buscará, nos dados, o padrão descrito.

Quais ferramentas e linguagens serão utilizadas, é algo que depende do contexto do processamento dos dados.

Elas podem ser:

- Recursos de busca e substituição de um editor de textos;

- Programas especializados disponíveis no sistema operacional;
- Estruturas, funções e construções de uma linguagem de programação;
- E até comandos internos e mecanismos do próprio shell.

Por falar em shell, que é o que nos interessa, os dados quase sempre virão de:

- Linhas de texto obtidas a partir de um arquivo;
- De um fluxo de dados (*entrada padrão*, ou *stdin*);
- De cadeias de caracteres (*strings*) associadas a parâmetros;
- Ou da listagem de nomes de arquivos e diretórios.

Para cada uma dessas origens, nós teremos inúmeras ferramentas à disposição, cada uma com suas características, aplicações e formas de receber e interpretar as descrições de padrões.

1.3 - Linguagens para descrever padrões

O problema é que não podemos dizer ao computador: *eu quero uma sequência de caracteres que inicia com "a", seguido de dois caracteres quaisquer e terminada com a sequência "c-a-t-e"*. Uma outra pessoa entenderia que você se refere a palavras como *"abacate"* ou *"alicate"*, mas o computador não fala a nossa língua: é preciso que exista um intérprete mediando comunicação através de uma linguagem em comum.

Qualquer programa interativo estabelece uma espécie de *"linguagem"* para que o utilizador possa especificar o que quer: podem ser cliques, gestos, opções na linha de comandos, etc… De certo modo, cada um com sua linguagem particular, esses programas são intérpretes das nossas vontades. Em sistemas operacionais *unix-like* (sistemas parecidos com o UNIX, como o GNU/Linux), o papel do intérprete é desempenhado por um programa muito especial: o ***shell***.

O shell é uma interface

O shell é um programa especialmente projetado para servir de ***interface*** entre nós, humanos, e o sistema operacional. Quando shell foi idealizado, decidiu-se que a

forma de comunicação entre o utilizador e o sistema operacional seria através de texto: tanto no sentido da construção de uma linguagem especial, quanto no sentido de que todos os dados transitariam entre o terminal e o computador na forma de fluxos de caracteres.

Observando a tabela de caracteres ASCII (`man ascii`, no GNU), nós vemos que ela tem 128 caracteres, mas apenas uma parte deles corresponde aos símbolos que usamos para escrever textos: todos os demais caracteres representam comandos e sinalizações para o terminal ou para o sistema operacional.

Textos são interfaces

Na ponta da interface que nos interessa, que é a expressão daquilo que queremos que o computador faça, todo um conjunto de palavras e símbolos especiais foi criado para que pudéssemos escrever nossos comandos de forma objetiva, inequívoca e abreviada. Isso resolveu o problema da comunicação de comandos, mas não demorou muito para que outro tipo de linguagem precisasse ser convencionada entre nós e o computador.

Em um sistema onde tudo fluia na forma de texto, no tempo em que livros e documentos estavam começando a ser transformados em arquivos digitais e os computadores tinham um poder computacional ligeiramente superior ao de qualquer calculadora de bolso dos nossos dias, algo precisava ser feito para acelerar a busca de dados com o menor consumo de recursos possível.

Foi neste contexto que surgiram as duas formas mais comuns de descrever padrões de texto: o **padrão de formação de nomes de arquivos** (*globs*) e as **expressões regulares** (*regex*, *regexp* ou *expreg*).

Um ponto importante sobre essas duas linguagens, é que elas não passam de **especificações** que, se não forem implementadas em um programa ou em uma linguagem de programação, não significam nada. Em outras palavras, *globs* e expressões regulares, sozinhas, não executam comandos e nem fazem buscas. Na prática, elas serão utilizadas como argumentos de comandos e funções: e **argumentos estão sempre no domínio de atuação do usuário**.

> ***Guarde bem:*** *globs e expressões regulares são ferramentas de comunicação para que nós, humanos, possamos descrever os padrões de texto que queremos que os programas localizem (ou ignorem) para nós.*

2 - Padrões de formação de nomes de arquivos

No GNU/Linux, ou em sistemas *unix-like* em geral, os **padrões de formação de nomes de arquivos** (*FNPM* ou *globs*, em inglês), são uma linguagem simplificada para descrever padrões de texto que casem com os nomes de arquivos em um diretório. Na quase totalidade dos shells, a presença dos caracteres especiais `*`, `?` e `[...]` na linha de um comando fará com que o shell proceda uma busca por nomes de arquivos que correspondam ao padrão descrito: é o mecanismo da expansão de nomes de arquivos, processado antes que a linha do comando seja efetivamente executada.

Esses caracteres são especiais porque vão além da sua função comum de símbolos gráficos, daí serem chamados de ***metacaracteres***.

2.1 - Metacaracteres

Nas descrições de padrões de nomes de arquivos, todos os metacaracteres, também chamados de *"curingas"*, **representam** caracteres válidos para ocupar determinada posição no texto buscado:

Metacaractere	Descrição
`*`	Zero ou mais caracteres quaisquer.
`?`	Um caractere qualquer.
`[...]`	Lista de caracteres válidos para uma posição.
`[!...]`	Lista de caracteres **inválidos** para uma posição.

> ***Importante!*** *Por padrão, o asterisco (`*`) e a interrogação (`?`) não casam com o caractere ponto (`.`) no início dos nomes de arquivos (arquivos ocultos).*

Qualquer coisa

Por exemplo, considere a listagem de diretório `~/tmp/regex`, abaixo:

```
:~/tmp/regex$ ls
classicos.mp3  contas.ods  inventario.ods  rock.mp3
clientes.ods   escola.ods  punk.mp3        wallpaper.png
compras.txt    foto.png    receita.txt
```

Se quisermos expandir apenas as planilhas (terminação `ods`), nós podemos descrever o padrão desta forma:

```
:~/tmp/regex$ echo *ods
```

É muito comum vermos o metacaractere `*` ser lido como *qualquer coisa*. Não está errado, contudo, já aqui, cabe uma dica muito importante:

> *Até que se torne uma segunda natureza, nós devemos sempre "traduzir" para o nosso idioma o que queremos dizer com o padrão. No caso do exemplo: zero ou mais caracteres quaisquer seguidos de `o`, `d` e `s`. Na verdade, será melhor ainda se dissermos o que queremos expressar enquanto estivermos digitando o padrão.*

Então, como uma prática para fixar a ideia e evitar dúvidas e enganos futuros, é importante que o metacaractere `*`, assim como os demais, seja lido segundo seu conceito exato: *zero ou mais caracteres quaisquer*.

Voltando ao comando do exemplo, quando teclarmos `Enter`, **antes do comando ser executado**, o shell tentará expandir `*ods`, comparando o padrão com os arquivos existentes no diretório `~/tmp/regex`. Se houver casamentos, os nomes encontrados serão incluídos na linha do comando como argumento do comando interno `echo`; caso contrário, se nada for encontrado, não haverá expansão alguma e o argumento do comando `echo` será `*ods`.

No nosso caso, o resultado será:

```
:~/tmp/regex$ echo *ods
clientes.ods contas.ods escola.ods inventario.ods
```

Por outro lado, se tentássemos expressar um padrão sem correspondência na lista de arquivos, o resultado seria:

```
:~/tmp/regex$ echo *odt
*odt
```

Ausência de quantificadores

Como não existem metacaracteres *quantificadores* na formação de nomes de arquivos, nós teremos que jogar com as quantidades implícitas de cada um dos metacaracteres:

- Zero ou mais: `*`
- Exatamente um: `?`
- Exatamente um da lista: `[...]`

Então, para especificar que queremos um número fixo de caracteres quaisquer, a alternativa será utilizar o metacaractere `?` quantas vezes forem necessárias para expressar essa quantidade.

Vamos dizer em voz alta a descrição do que queremos descrever no próximo exemplo: *exatamente quatro caracteres quaisquer seguidos do caractere ponto e zero ou mais caracteres quaisquer*, o que pode ser expresso com o *glob*:

```
????.*
```

Passando isso para o shell:

```
:~/tmp/regex$ echo ????.*
foto.png punk.mp3 rock.mp3
```

Casamentos alternativos

A lista é usada para especificar os caracteres que poderão ocupar uma determinada posição no padrão. A ideia é que o caractere **tem que existir** e só pode ser um dos especificados na lista.

Por exemplo, *nomes iniciados com o caractere* `c` *seguido de* `l` ***ou*** `o` *e de zero ou mais caracteres quaisquer*:

```
c[lo]*
```

Executando no shell:

```
:~/tmp/regex$ echo c[lo]*
classicos.mp3 clientes.ods compras.txt contas.ods
↑             ↑            ↑           ↑
```

Em vez disso, se quisermos especificar os caracteres **proibidos** em determinada posição, nós podemos *negar* a lista. Por exemplo: *nomes iniciados com o caractere* `r` *seguidos de zero ou mais caracteres quaisquer*.

```
:~/tmp/regex$ ls r*
receita.txt  rock.mp3
```

Se quisermos excluir `receita.txt` do resultado da listagem, uma das nossas opções é negar o segundo caractere (`e`), expressando que queremos um caracteres qualquer na segunda posição, exceto o caractere `e`.

Como resultado:

```
:~/tmp/regex $ ls r[!e]*
rock.mp3
```

2.2 - Outros usos no shell (Bash)

No shell Bash, os padrões de formação de nomes de arquivos também podem ser utilizados para o casamento com *strings*:

- No comando composto `[[` com os operadores `==` e `!=`;
- Nas cláusulas da estrutura `case`;
- Em todas as expansões de parâmetros que envolvam a descrição de padrões.

Em nenhum desses casos de uso acontecerá a expansão de nomes de arquivos: ou seja, trata-se apenas de descrição de um padrão de texto utilizando os mesmos metacaracteres.

No 'test do Bash'

O comando composto `[[`, com o operador `==`, avalia a afirmação de que a string da esquerda corresponde ao padrão descrito na direita. Por exemplo:

```
[[ abacaxi == a*i ]] && echo casou || echo não casou
```

Aqui, a *string* literal `abacaxi` é comparada com um padrão descrito como *uma string iniciada com o caractere `a`, seguido de zero ou mais caracteres quaisquer e terminada com o caractere `i`*. A saída da linha dos comandos será:

```
casou
```

Além de `abacaxi`, o mesmo padrão casaria com `aqui`, `abri`, `acertei`, `ai`, `'agora eu fui'`, etc…

No 'case'

Na estrutura `case`, os globs funcionam da mesma forma:

```
str=banana
case $str in
```

```
 a*) echo a string começa com 'a';;
 b*) echo a string começa com 'b';;
  *) echo a string não começa nem com 'a' e nem com 'b'
esac
```

No exemplo, a expansão da variável `str` (que, no momento, tem o valor `banana`) é comparada com os padrões descritos nas cláusulas:

- `a*` - Caractere `a` seguido de zero ou mais caracteres quaisquer.
- `b*` - Caractere `b` seguido de zero ou mais caracteres quaisquer.
- `*` - Zero ou mais caracteres quaisquer.

Havendo casamento, o comando correspondente é executado. Como a variável `$str` expande `banana`, o casamento ocorre com o padrão `b*`, o que resulta na execução do comando que imprime:

```
a string começa com b
```

O `case` ainda conta com um curinga a mais, a barra vertical (`|`), que expressa ***padrões alternativos***.

Por exemplo:

```
case $str in
    banana|laranja) echo 'É fruta';;
    zebra|tigre)    echo 'É bicho';;
esac
```

Nas expansões de parâmetros

O Bash pode transformar as expansões de parâmetros de muitas formas: algumas delas, de acordo com a descrições de padrões que serão comparados com as strings associadas a esses parâmetros.

Observe:

```
:~$ var='https://blauaraujo.com'
:~$ echo ${var#*//}
blauaraujo.com
```

Aqui, o padrão `*//` (/zero ou mais caracteres quaisquer seguidos dos caracteres `//`) foi comparado com o início da string associada a `var` e, tendo sido encontrado, o trecho correspondente ao padrão foi removido da expansão (só da expansão, não da variável).

2.3 - Configurações do Bash para globbing

O Bash é um shell que possui opções para quase todo tipo de comportamento. Por exemplo, na tabela dos metacaracteres, nós vimos que o asterisco (`*`) não casa com o caractere ponto (`.`) se ele aparecer no começo do nome de um arquivo:

```
:~/tmp/regex $ ls -a
.  ..  exemplos1  banana  carambola  .manga
:~/tmp/regex $ echo *
banana carambola exemplos1
```

Este é o comportamento padrão na quase totalidade dos sistemas que utilizam o Bash, mas é um comportamento que pode ser alterado manipulando as opções do shell. Para cuidar isso, existe o comando interno `shopt`.

O comando 'shopt'

Passando apenas o nome da opção como argumento, nós podemos ver seu estado atual. Por exemplo, para conferir o estado da opção `dotglob`, que é justamente a opção que determina se o asterisco casará ou não com o ponto no começo do nome do arquivo:

```
:~/tmp/regex $ shopt dotglob
dotglob         off
```

Para habilitar uma opção, nós utilizamos o argumento `-s` (de *set*):

```
:~/tmp/regex $ shopt -s dotglob
:~/tmp/regex $ shopt dotglob
dotglob          on
```

Agora, que `dotglob` está habilitado, vejamos o que acontece com a expansão do asterisco:

```
:~/tmp/regex $ echo *
banana carambola exemplos1 .manga
```

Curiosamente, mesmo casando com o ponto inicial de `.manga`, repare que não houve casamento com os dois nomes especiais `.` (diretório corrente) `..` (diretório pai)! Para restaurar a condição padrão, nós podemos encerrar a sessão do shell ou executar `shopt` com o argumento `-u` (de *unset*):

```
:~/tmp/regex $ shopt -u dotglob
:~/tmp/regex $ shopt dotglob
dotglob          off
:~/tmp/regex $ echo *
banana carambola exemplos1
```

Mas, atenção! Mesmo com `dotglob` desabilitado, o metacaractere `*` não tem a restrição do ponto quando estamos casando ***strings***!

Observe:

```
:~/tmp/regex $ shopt dotglob
dotglob            off
:~/tmp/regex $ var=.com
:~/tmp/regex $ [[ $var == * ]] && echo casa || echo não casa
casa
```

As opções podem ser diferentes em cada modo

Algumas opções do Bash podem ser configuradas de formas diferentes para o **modo interativo** (aquele que utilizamos pelo terminal) e o **modo não-interativo** (quando shell é executado em scripts ou pela invocação do comando `bash -c`). Por exemplo:

```
# Modo não-interativo
:~/tmp/regex $ bash -c "shopt extglob"
extglob             off

# Modo interativo
:~/tmp/regex $ shopt extglob
extglob             on
```

É importante saber disso porque, muitas vezes, nós testamos comportamentos do shell no terminal, mas ficamos completamente perdidos quando o mesmo procedimento não apresenta os mesmos resultados nos nossos scripts.

Opções relativas aos casamentos de nomes de arquivos

A tabela abaixo mostra as opções do Bash que podem afetar o resultado de casamentos com padrões de formação de nomes de arquivos:

Opção	Descrição	INT	NINT
`dotglob`	Permite que o casamento de padrões com nomes de arquivos iniciados com ponto (arquivos ocultos) sejam expandidos.	Desligada	Desligada
`extglob`	Permite o uso de sintaxes adicionais para casamento de padrões.	Ligada	Desligada

`failglob`	Faz com que padrões sem correspondência gerem uma mensagem de **erro de expansão**.	Desligada	Desligada
`globasciiranges`	Define que o conjunto de caracteres que irão corresponder às faixas definidas listas (`[...]`) seguirão a tabela ASCII em vez dos caracteres do idioma local.	Ligada	Ligada
`globstar`	Permite o uso de dois asteriscos (`**`) para incluir casamentos com nomes de diretórios e subdiretórios, ou apenas de diretórios e subdiretórios, se vier seguido da barra (`/`).	Desligada	Desligada
`nocaseglob`	Permite casar padrões independente dos caracteres estarem em caixa alta ou baixa.	Desligada	Desligada
`nullglob`	Expande uma string nula (vazia), em vez do próprio padrão, quando não há casamento com nomes de arquivos.	Desligada	Desligada
`noglob`	Se habilitada, não permite expansões de nomes de arquivos (opção gerenciada com o comando `set -o`).	Desligado	Desligado

A variável 'GLOBIGNORE'

Além das opções do shell, nós ainda podemos trabalhar com a variável `GLOBIGNORE` para alterar a os resultados expandidos. Com ela, nós podemos listar os padrões que representarão os casamentos que devem ser **removidos** do resultado da expansão. Por exemplo:

```
:~/tmp/regex $ echo e*
empresa.ods  encontro.png  escola.txt
```

Se quisermos ignorar as expansões de nomes terminados em `png`:

```
:~/tmp/regex $ GLOBIGNORE=*png
:~/tmp/regex $ echo e*
empresa.ods  escola.txt
```

Ou ainda, separando os padrões com dois pontos (`:`)…

```
:~/tmp/regex $ GLOBIGNORE=*png:*txt
:~/tmp/regex $ echo e*
empresa.ods
```

Também é possível ignorar padrões de nomes de arquivos com a opção `extglob`, como veremos a seguir, mas a variável especial `GLOBIGNORE` é interessante pela possibilidade que oferece de listarmos vários padrões que queremos ignorar sem a alteração das configurações do shell.

2.4 - Globs estendidos

Quando habilitados, os globs estendidos ampliam em muito a nossa capacidade de representar padrões mais complexos.

Veja na tabela abaixo:

Sintaxe	Descrição
`?(LISTA_DE_PADRÕES)`	Casa com **zero ou uma** ocorrência dos padrões na lista.
`*(LISTA_DE_PADRÕES)`	Casa com **zero ou mais** ocorrências dos padrões na lista.
`+(LISTA_DE_PADRÕES)`	Casa com **uma ou mais** ocorrências dos padrões na lista.
`@(LISTA_DE_PADRÕES)`	Casa com apenas um dos padrões na lista.
`!(LISTA_DE_PADRÕES)`	Casa com tudo, menos os padrões na lista.

> *A separação dos padrões alternativos é feita com a barra vertical (`|`).*

Se repararmos bem, nós veremos que os globs estendidos trazem exatamente os dois elementos sintáticos que faltavam para que pudéssemos descrever praticamente qualquer padrão sem recorrer a expressões regulares: especificações de quantidades e padrões alternativos.

Observe:

```
:~/tmp/regex $ echo *+(o|c)?(s|k).mp3
classicos.mp3 rock.mp3
```

Qual foi o padrão descrito?

```
+---------- Zero ou mais caracteres
|    +------- Seguidos de uma ou mais ocorrências de 'o' ou 'c'
|    |      +-- Seguidos de zero ou uma ocorrência de 's' ou 'k'
|    |      |     +-- Seguidos de '.', 'm', 'p' e '3'
|    |      |     |
↓    ↓      ↓     ↓
* +(o|c) ?(s|k) .mp3
```

Equivalências com expressões regulares

Nós ainda não estudamos as expressões regulares, mas fica o registro para ser entendido mais adiante:

```
EXTGLOB              REGEX       DESCRIÇÃO
-------------------------------------------------------------
?(LISTA_DE_PADRÕES)  (...|...)?  Zero ou uma ocorrências
*(LISTA_DE_PADRÕES)  (...|...)*  Zero ou mais ocorrências
+(LISTA_DE_PADRÕES)  (...|...)+  Uma ou mas ocorrências
@(LISTA_DE_PADRÕES)  (...|...)   Apenas uma ocorrência
!(LISTA_DE_PADRÕES)     ---      Sem equivalente direto
```

Casamento com strings

Os globs estendidos também podem ser usados para casamentos com strings no Bash. Por exemplo, numa estrutura `case`:

```
#!/bin/bash

shopt -s extglob

case $1 in
        !(lona|cara)) echo "Não é 'lona' nem 'cara'..." ;;&
   @(casa|mesa|tela)) echo "Casou com $1!" ;;
                  *) echo "não casou com nada esperado!"
esac
```

Detalhes importantes:

- Estando no modo **não-interativo**, temos que habilitar `extglob`.
- O operador `;;&` diz que as cláusulas seguintes também devem ser testadas.
- Os padrões do `case` **não são** condições para execução das cláusulas!
- O que está sendo testado é o casamento dos padrões com a expansão.

Ou então, com o comando composto `[[`:

```
#!/bin/bash

shopt -s extglob
read -p 'De que estilo musical você gosta? '

if [[ "$REPLY" == *+(rock|punk|metal)*  ]]; then
   echo 'Hell, yeah!'
elif [[ $REPLY == *+(pagode|samba)* ]]; then
   echo 'Gostei da ginga!'
else
   echo 'Por que você não gosta de música de verdade? :-('
fi
```

Repare que nós podemos combinar os metacaracteres básicos com as formas estendidas. No exemplo, essa combinação está sendo usada para casar com a ocorrência obrigatória de pelo menos um dos padrões em qualquer parte da string expandida.

3 - Expressões regulares

Expressões regulares (ou *regex*, de ***REG****ular* ***EX****pressions*, como são popularmente conhecidas) são expressões que nos permitem descrever, com bastante detalhes, praticamente qualquer padrão de texto.

3.1 - Metacaracteres

Os metacaracteres são, como vimos, caracteres que vão além de seu papel normal de símbolos gráficos em um texto. No contexto da construção de expressões regulares, isso significa que alguns caracteres assumirão algum papel especial, que pode ser de:

- Representar a ocorrência de caracteres (representantes e classes);
- Descrever a quantidade em que esses caracteres ocorrem (quantificadores);
- Especificar com que parte do texto o padrão deve coincidir (âncoras);
- Alterar a precedência, a associação e a relação entre os padrões presentes na expressão (operadores diversos);
- Transformar metacaracteres em caracteres normais e vice-versa (escape).

A tabela abaixo mostra os metacaracteres comuns à maioria das implementações de expressões regulares para uso no shell de sistemas **unix-like** e GNU:

Metacaractere	Tipo	Descrição
`.`	Representante	Qualquer caractere único.
`[...]`	Representante	Um caractere na lista.

`[^...]`	Representante	Um caractere que não esteja na lista.
`\w`	Classe	Um caractere válido em palavras: `[A-Za-z0-9_]`
`\W`	Classe	Um caractere inválido em palavras: `[^A-Za-z0-9_]`
`\s`	Classe	Um caractere em branco.
`\S`	Classe	Tudo menos caracteres em branco.
`?`	Quantificador	A entidade anterior (caractere literal, padrão ou representante) pode ocorrer zero ou uma vez.
`*`	Quantificador	A entidade anterior pode ocorrer zero ou mais vezes.
`+`	Quantificador	A entidade anterior pode ocorrer uma ou mais vezes.
`{min,max}`	Quantificador	A entidade anterior pode ocorrer em qualquer quantidade entre `mín` e `max` (inclusive).
`^`	Âncora	Marca o começo de uma linha.
`$`	Âncora	Marca o fim de uma linha.
`\b`	Âncora	Especifica que o casamento deve ocorrer no início ou no fim de uma palavra.
`\B`	Âncora	Especifica que o casamento **não deve** ocorrer no início ou no fim de uma palavra.
`\`	Escape	Remove o significado especial de metacaracteres (ou atribui significado especial a caracteres).
`|`	Operador	Expressa padrões alternativos (um ou outro).

`(...)`	Grupo	Agrupa, aninha e define a precedência e a associação de padrões.
`\1...\9`	Retrovisor	Reutiliza padrões casados em grupos anteriores conforme a ordem das ocorrências (máximo de 9).

Não confunda!

Os metacaracteres `?` e `*` não devem ser confundidos com aqueles que vimos na formação de padrões de nomes de arquivos. A principal diferença é que, nas expressões regulares as funções de representar caracteres e descrever suas quantidades são atribuídas a metacaracteres diferentes, ao passo que, nos *globs*, os metacaracteres acumulam as duas funções. Para fazermos um paralelo entre as duas linguagens, observe a tabela abaixo:

Glob	Regex	Descrição
`*`	`.*`	Zero ou mais caracteres quaisquer.
`?`	`.`	Exatamente um caractere qualquer.
`[...]`	`[...]`	Exatamente um caractere qualquer da lista.
`[!...]`	`[^...]`	Um caractere que não esteja na lista.

3.2 - O metacaractere de escape

Embora seja quase sempre lembrado por remover os "poderes especiais" de metacaracteres (uma *kriptonita*, como diz Aurélio Jargas), a barra invertida também faz com que alguns caracteres normais ganhem "poderes especiais", como vimos em algumas classes de caracteres (`\s`, `\w`, etc…), nas âncoras das bordas (`\b` e `\B`) e com os retrovisores (`\1...\9`).

Algumas implementações de expressões regulares em programas também podem

exigir o escape de caracteres para que eles se comportem como metacaracteres.

Observe:

```
:~$ var='2 5 10'
:~$ grep '[0-9]' <<< $var
[2] [5] [10]
```

> *Os trechos entre colchetes na saída são os casamentos encontrados pelo `grep` que, no terminal, apareceriam em destaque.*

Se quisermos que o casamento ocorra apenas com dois dígitos numéricos seguidos, nós podemos utilizar o quantificador `{2}` (*exatamente duas ocorrências da entidade anterior*):

```
:~$ var='2 5 10'
:~$ grep '[0-9]{2}' <<< $var
:~$
```

Aparentemente, nenhum casamento foi encontrado, mas isso é um *"falso negativo"*, como podemos confirmar escapando as chaves:

```
:~$ var='2 5 10'
:~$ grep '[0-9]\{2\}' <<< $var
2 5 [10]
```

Esse comportamento pode ser alterado por opções explícitas do `grep`, como a opção `-E`:

```
:~$ var='2 5 10'
:~$ grep -E '[0-9]{2}' <<< $var
2 5 [10]
```

É justamente isso que caracteriza dois dos modos de expressões regulares do `grep`: os modos **básico** (BRE) e **estendido** (ERE). No modo BRE, os metacaracteres `?`,

`+`, `{`, `|`, `(` e `)` perdem seu significado especial, a menos que sejam escapados ou que o modo ERE seja habilitado com a opção `-E`. Aliás, o utilitário `sed` também trabalha com expressões regulares básicas e estendidas sob as mesmas condições.

3.3 - Representantes

Os metacaracteres **representantes** são aqueles que, como o nome diz, representam a ocorrência de um (e apenas um) caractere. Nesta categoria encontram-se:

Metacaractere	Descrição
`.`	Qualquer caractere único.
`[...]`	Um caractere na lista.
`[^...]`	Um caractere fora da lista.

> *Uma nota importante, antes de entrarmos nos próximos exemplos: o `grep` sempre retorna **linhas inteiras** que contenham o padrão descrito na expressão regular. Para obter apenas as partes das linhas que casarem com o padrão, é preciso informar a opção `-o` (**only**). Deste modo, cada correspondência encontrada será exibida em uma linha na saída.*

O ponto

O ponto casa com exatamente um caractere qualquer:

```
:~$ var='banana bacana barata bandana'
:~$ grep -o 'ba.a.a' <<< $var
banana
bacana
barata
```

Ele também casa com espaços e o próprio ponto:

```
:~$ num='2.75 2,75 2 75'
:~$ grep -o '2.75' <<< $num
2.75
2,75
2 75
```

Quando precisarmos representar um caractere ponto textual, ele tem que ser escapado:

```
:~$ num='2.75 2,75 2 75'
:~$ grep -o '2\.75' <<< $num
2.75
```

Mas o ponto **não casa** com o caractere de fim de linha:

```
:~$ var=$'1234\n'
:~$ grep '....' <<< "$var"
1234
:~$ grep '.....' <<< "$var"
:~$
```

No exemplo, nós utilizamos uma expansão de caracteres ANSI-C (`$'...'`) para que a quebra de linha (`\n`) fosse acrescentada como um quinto caractere à **string** atribuída a `var`.

Quando descrevemos o padrão buscado com quatro pontos, o `grep` nos retornou uma linha contendo os 4 primeiros caracteres em `var`. Mas, quando tentamos descrever um quinto caractere qualquer no padrão, nada foi retornado: o padrão simplesmente não tem correspondência na linha recebida pelo `grep`.

Desafio!

Se o que acabamos de dizer está correto, como você explica a saída abaixo?

```
:~$ var=$'1234\n'
:~$ grep '.*' <<< "$var"
1234
```

```
:~$
```

Lista

A lista só permite o casamento com caracteres descritos entre os colchetes:

```
:~$ grep -o 'b.ta' <<< $'bata beta bota b\tta'
bata
beta
bota
b       ta
```

Aqui, com o ponto, o segundo caractere pode ser, literalmente, qualquer coisa, inclusive caracteres como a tabulação na quarta palavra. Pode ser que isso seja desejável em algumas situações, mas é sempre bom avaliar se não podemos ser mais específicos.

Observe o exemplo abaixo:

```
:~$ grep -o 'b[ao]ta' <<< $'bata beta bota b\tta'
bata
bota
```

Desta vez, nós deixamos claro que queremos apenas um de dois caracteres na segunda posição (`a` ou `o`), o que forma um padrão que casa com apenas duas palavras: `bata` ou `bota`.

Sem significados especiais

Dentro dos colchetes de uma lista, todos os caracteres são textuais, menos o circunflexo (`^`) **quando ele for o primeiro caractere da lista**: ou seja, se quisermos casar com um caractere circunflexo, ele não poderá ser o primeiro a aparecer na lista.

Por exemplo:

```
:~$ grep -o '2[.^*$]75' <<< '2.75 2^75 2*75 2$75'
2.75
2^75
2*75
2$75
```

Casando colchetes

Os colchetes também podem entrar na lista, mas é preciso tomar cuidado com o caractere `]`: se for o caso, ele tem que ser o **primeiro caractere na lista**:

```
:~$ grep -o '2[].,[]75' <<< '2.75 2,75 2[75 2]75'
2.75
2,75
2[75
2]75
```

Faixas de caracteres

Outra coisa que podemos fazer nas listas é descrever faixas de caracteres. Embora seja mais comum o uso de faixas de letras e números, é possível criar faixas com quase todos os tipos de caracteres iniciais e finais. A única limitação (e um bom motivo para não abusar das possibilidades) é que os caracteres válidos dependerão das coleções de caracteres da localidade do sistema:

```
:~$ grep -o '[9-a]' <<< '9 ; : < = > ? @ a'
9
a
:~$ LC_ALL=C grep -o '[9-a]' <<< '9 ; : < = > ? @ a'
9
;
:
<
=
>
?
```

```
@
a
```

Na tabela de caracteres da localidade `pt_BR` (padrão do meu sistema), os símbolos gráficos da string testada não estão entre `9` e `a`. Mas, alterando a localidade para `C`, a tabela ASCII é utilizada e todos os símbolos gráficos casam com o intervalo da lista.

Os intervalos mais comuns são:

- `A-Z` - letras maiúsculas
- `a-z` - letras minúsculas
- `0-9` - dígitos

> ***Importante!*** *Quando quiser casar caracteres maiúsculos e minúsculos, especifique sempre a faixa dos maiúsculos primeiro:* `[A-Za-z]`.

Também não precisamos utilizar sempre os extremos dos intervalos. Faixas como `b-f` ou `3-8`, por exemplo, são perfeitamente válidas.

Casando o traço

Como o traço (`-`) é utilizado para expressar faixas de caracteres, ele também exigirá um cuidado especial quando for apenas um caractere descrito na lista. Nesta situação, ele deverá ser o primeiro ou o último!

```
:~$ grep -o '[0-9-]' <<< '1 2 3 -'
1
2
3
-
```

É mais comum vermos o traço sendo utilizado na última posição da lista, mas isso não é uma regra.

Lista negada

A lista negada segue o mesmo conceito de uma lista normal, a diferença é que, neste caso, a leitura que se faz é *"tudo, menos o que está listado"*.

Observe o exemplo:

```
:~$ grep -o '[0-9]' <<< '1 2 3 a b c'
1
2
3

# Negando os números...
:~$ grep -o '[^0-9]' <<< '1 2 3 a b c'



a

b

c
```

Outro desafio!

Consegue explicar as linhas vazias desta saída?

3.4 - Quantificadores

Por padrão, cada entidade (um caractere textual, um representante ou um padrão agrupado) é tratada como uma **ocorrência única e obrigatória** no padrão. Para alterar isso, nós temos os metacaracteres quantificadores:

Metacaractere	Descrição
`?`	A entidade anterior pode ocorrer zero ou uma vez.

`*`	A entidade anterior pode ocorrer zero ou mais vezes.
`+`	A entidade anterior pode ocorrer uma ou mais vezes.
`{min,max}`	A entidade anterior pode ocorrer em qualquer quantidade entre o `min` e `max`.
`{0,max}`	A entidade anterior pode ocorrer de zero até `max` vezes.
`{min,}`	A entidade anterior pode ocorrer pelo menos `min` vezes.
`{qtde}`	Especifica a quantidade exata de ocorrências da entidade anterior.

3.5 - Âncoras

As âncoras não casam com caracteres, mas nos ajudam a especificar a posição da ocorrência de um padrão:

Metacaractere	Descrição
`^`	Marca o começo de uma linha.
`$`	Marca o fim de uma linha.
`\b`	Especifica que o padrão ocorre no início ou no fim de uma palavra.
`\B`	Especifica que o padrão **não ocorre** no início ou no fim de uma palavra.

> *No contexto das expressões regulares, "palavras" são sequências de caracteres formadas apenas por letras maiúsculas e minúsculas, dígitos e o caractere sublinhado:* `[A-Za-z0-9_]`.

3.6 - Classes

As classes são conjuntos predefinidos de caracteres válidos ou proibidos em dada posição do padrão. As classes mais comuns são:

Classe	Lista	Descrição
`\w`	`[A-Za-z0-9_]`	Caracteres válidos em palavras.
`\W`	`[^A-Za-z0-9_]`	Caracteres inválidos em palavras.
`\s`	`[ \t\n\r\f\v]`	Espaço ou tabulação.
`\S`	`[^ \t\n\r\f\v]`	Tudo menos espaço ou tabulação.

> *Na literatura estrangeira, as classes de caracteres precedidas pela contrabarra também são chamadas de "operadores".*

Dependendo da implementação das expressões regulares nos programas, nós ainda podemos encontrar outras classes, como:

- `\d`: dígitos decimais
- `\x`: dígitos hexadecimais
- `\O`: dígitos octais

Contudo, por não estarem universalmente disponíveis no shell GNU, seu uso deve ser evitado em favor das classes conhecidas como **classes POSIX**.

3.7 - Classes POSIX

As classes POSIX também são conjuntos predefinidos de caracteres, mas, embora sejam representadas entre colchetes, elas não funcionam fora de listas.

Por exemplo:

```
:~$ grep '\w ' <<< 'Árvore Banana CASA'
Árvor[e] Banan[a] CASA
```

Aqui, a classe `\w` pôde ser utilizada tranquilamente fora de uma lista, mas teria problemas em listas, porque a contrabarra perderia seu significado especial:

```
:~$ grep '\sB' <<< 'Árvore Banana CASA'
Árvore[ B]anana CASA
:~$ grep '[\s]B' <<< 'Árvore Banana CASA'
:~$
```

Com as classes POSIX, ocorre o contrário:

```
:~$ grep '[:space:]B' <<< 'Árvore Banana CASA'
grep: a sintaxe de categoria de caracteres é [[:space:]], e não
[:space:]
```

Destaque especial para a mensagem de erro que pode levar a conclusões equivocadas! O que a mensagem chama de *"sintaxe de categoria de caracteres"* é, na verdade, a sintaxe de uma lista contendo uma classe POSIX (uma *classe nomeada*, no caso):

```
:~$ grep '[[:space:]]B' <<< 'Árvore Banana CASA'
Árvore[ B]anana CASA
```

Ou ainda, para deixar mais claro…

```
:~$ grep '[[:upper:]B]a' <<< 'Tamanduá Banana laranja'
[Ta]manduá [Ba]nana laranja
```

Classes nomeadas

Quando dizemos "classes POSIX", geralmente estamos nos referindo a apenas uma das definições das normas POSIX para classes de caracteres, as chamadas **classes nomeadas**:

Classe	Lista	Descrição
`[:upper:]`	`[A-Z]`	Letras maiúsculas.
`[:lower:]`	`[a-z]`	Letras minúsculas.
`[:alpha:]`	`[A-Za-z]`	Maiúsculas/minúsculas.
`[:alnum:]`	`[A-Za-z0-9]`	Letras e números.
`[:word:]`	`[A-Za-z0-9_]`	Letras, números e sublinhado.
`[:digit:]`	`[0-9]`	Dígitos decimais.
`[:xdigit:]`	`[0-9A-Fa-f]`	Dígitos em hexadecimal.
`[:blank:]`	`[ \t]`	Espaço e tabulação.
`[:space:]`	`[ \t\n\r\f\v]`	Caracteres em branco.
`[:graph:]`	`[^ \t\n\r\f\v]`	Caracteres imprimíveis.
`[:print:]`	`[^\t\n\r\f\v]`	Imprimíveis e o espaço.
`[:punct:]`	(uma lista enorme!)	Pontos e símbolos gráficos.

Contudo, ainda existem outras definições de classes POSIX, como os *elementos de coleção* e as *classes de equivalência*.

Elementos de coleção

Podem ser:

- Um caractere;
- Uma sequência de caracteres que conte como um caractere único;
- Ou um nome predefinido para uma sequência de caracteres.

Quando os elementos de uma coleção aparecem entre `[.` e `.]`, tudo que estiver entre os pontos será tratado como uma sequência textual do padrão entre as opções da lista. Sendo assim, uma coleção `[.ch.]`, onde `ch` é visto como apenas um caractere no idioma tcheco, por exemplo, casaria com as ocorrências do caractere

`c` seguido do caractere `h` na string. Mas, por ser altamente dependente da localização, este recurso raramente é usado para descrever padrões no nosso idioma ou na localização padrão `C`, especialmente para avaliações feitas com utilitários GNU.

Classes de equivalência

As localizações POSIX podem trazer especificações de que alguns caracteres serão considerados idênticos para efeito de ordenação. É o caso de caracteres acentuados, que poderão ser ignorados na ordenação de uma lista de palavras:

```
:~$ sort -d <<< $'árvore\nacento\nátomo\npeixe\nseta'
acento
peixe
árvore
seta
átomo
```

Perceba que, com a opção `-d` o utilitário `sort` ignorou o caractere `á`, de `árvore` e `átomo`, procedendo a ordenação levando em conta o segundo caractere. No contexto de uma expressão regular, as classes de equivalência dirão que o caractere `a`, por exemplo, e suas versões acentuadas deverão ser levados em conta no casamento:

```
:~$  grep '[[=a=]]' <<< 'árvore acento átomo peixe seta'
[á]rvore [a]cento [á]tomo peixe set[a]
```

3.8 - Operador 'ou'

O operador "ou", representado pela barra vertical (`|`), expressa **padrões** alternativos, ou seja, os padrões à esquerda e à direita da barra são diferentes, mas ambos são válidos para efeito de um casamento:

```
:~$ str=$'bom dia\nboa tarde\nboa noite\ndia e noite'
:~$ grep -E 'bom dia|boa' <<< "$str"
[bom dia]
[boa] tarde
[boa] noite
```

Repare que o casamento foi feito com os padrões alternativos `bom dia` e `boa`, não com `dia` ou `boa`. Isso significa que o operador `|` trabalha com padrões inteiros antes e depois dele, não com partes de padrões. Se quisermos delimitar os padrões alternativos, será preciso recorrer aos agrupamentos, por exemplo:

```
:~$ str=$'bom dia\nboa tarde\nboa noite'
:~$ grep -E 'bom dia|boa (tarde|noite)' <<< "$str"
bom dia
boa tarde
boa noite
```

3.9 - Grupos

Os parêntesis `(...)` são utilizados para agrupar subpadrões. Sendo assim, no exemplo anterior, o grupo com os padrões alternativos `tarde` ou `noite` ainda são parte do padrão iniciado com `boa-espaço`. Consequentemente, o primeiro operador `|` da nossa expressão está lidando com os padrões alternativos `bom dia` e `boa (tarde|noite)`.

Referências prévias (retrovisores)

O mecanismo das expressões regulares faz um registro de todos os trechos casados com os subpadrões na ordem em que são encontrados.

Então, ainda no exemplo anterior, a string casada com o padrão `(tarde|noite)` ficará registrada e poderá ser referenciada na expressão regular pelo operador numerado `\1`:

```
:~$ str=$'bom dia\nboa noite tarde\nboa noite noite'
:~$ grep -E 'boa (tarde|noite) \1' <<< "$str"
boa noite noite
```

4 - Casamentos gulosos e PCRE

As implementações de expressões regulares POSIX e GNU nos modos básico (BRE) e estendido (ERE) sempre tentarão casar o padrão com a maior sequência de caracteres possível. Esse comportamento é chamado de casamento *ganancioso* (*greedy*, em inglês) ou, como Aurélio Jargas e Julio Neves costumam dizer: *"guloso"*.

Normalmente, isso não afeta a maior parte dos usos das expressões regulares: localizar linhas de texto inteiras que casem com o padrão, por exemplo. Contudo, quando se trata de extrair e processar apenas a parte casada, isso pode ser um pouco mais complicado, mas sempre haverá uma solução!

Antes de encontrar soluções, o problema precisa ser entendido. Então, vamos observar um exemplo com o arquivo `/etc/passwd`. Aqui, o nosso objetivo é extrair apenas os três primeiros campos da linha iniciada com a string `blau`:

```
:~$ grep -Eo '^blau:(.*:){2}' /etc/passwd
blau:x:1000:1000:Blau Araujo,,,:/home/blau:
```

4.1 - Resolvendo casamentos gulosos

Como podemos ver, a parte casada da string incluiu todos os seis primeiros campos, o que aconteceu porque `.*:` sempre casará com a maior sequência de caracteres seguida de `:` na linha. Isso significa que, nos modos ERE e BRE, se quisermos a parte da string que casa com o padrão, nós temos algumas opções…

Solução 1: regex mais verbosa

Um opção é tentamos ser mais verbosos e específicos nas nossas expressões regulares:

```
:~$ sed -nE 's/^(blau:)((.*:){2})(.*:){3}.*/\1\2/p' /etc/passwd
blau:x:1000:
```

Solução 2: trocar a regex por recursos do shell

Também podemos abandonar a abordagem por expressões regulares para lançar mão de outros recursos do shell para processar os casamentos encontrados:

```
:~$ grep '^blau:' /etc/passwd | cut -d':' -f1-3
blau:x:1000
```

Solução 3: usar a cabeça

Outra solução, mais engenhosa, é negar os caracteres de separação:

```
:~$ sed -nE 's/^(blau:([^:]+:){2}).*/\1/p' /etc/passwd
blau:x:1000:
```

Ou, com o `grep`:

```
:~$ grep -Eo '^blau:([^:]+:){2}' /etc/passwd
blau:x:1000:
```

Solução 4: utilizar outra especificação de regex

Felizmente, o `sed` e o `grep` não são os únicos utilitários que trabalham com regex na linha de comandos. Nós temos, por exemplo, o interpretador da linguagem Perl, que conta com especificações de regex bem mais avançadas.

No no nosso caso, uma solução com o `perl` poderia ser assim:

```
:~$ perl -ne 'print if s/^(blau:(.*?:){2}).*/\1/p' /etc/passwd
blau:x:1000:
```

Mas, repare nesta construção:

```
.*?
```

Pelo que vimos sobre os metacaracteres, tanto `*` quanto `?` são quantificadores: por que estamos utilizando dois quantificadores para a mesma entidade, afinal?

Isso é o que chamamos de **quantificadores dobrados**, como veremos melhor a seguir, e é uma das especificações de expressões regulares compatíveis com o Perl (PCRE). O tipo de quantificador dobrado que utilizamos informa ao mecanismo de processamento da regex que o casamento deve ser feito com a menor quantidade de caracteres possível, anulando o casamento guloso.

4.2 - Quantificadores dobrados

Em meados dos anos 1990, a versão 5 da linguagem Perl introduziu uma série de novos operadores de expressões regulares, entre eles, o *modificador de quantificadores*, também chamados de *quantificadores dobrados*, que visa justamente implementar a possibilidade de realizar um casamento com a mínima sequência de caracteres que corresponda ao padrão. Então, nas especificações do Perl 5, todo quantificador seguido de `?` indicará um casamento *"não-guloso"*:

Modificador	Descrição
`??`	Zero ou um "não-guloso".
`*?`	Zero ou mais "não-guloso".
`+?`	Um ou mais "não-guloso".
`{min,max}?`	De mínimo a máximo "não-guloso".

Os quantificadores dobrados não estão implementados nos modos BRE e ERE, adotado por ferramentas GNU como o `awk`, o `grep` e o `sed`, mas o `grep` tem um modo parcialmente implementado de compatibilidade com as especificações Perl (PCRE) com a opção `-P`.

Portanto, isso é possível:

```
:~$ grep -Po '^blau:(.*?:){2}' /etc/passwd
blau:x:1000:
```

4.3 - Operadores estendidos do Perl

Ainda entre as novidades do Perl 5, agora nós temos uma série de operadores capazes de dotar as expressões regulares de recursos que, originalmente, ficariam para os comandos e utilitários do shell encarregados pelo pós-processamento dos resultados obtidos.

De modo geral, esses *operadores estendidos* se parecem muito com agrupamentos (e, de fato, eles agrupam). A principal diferença, porém, é que o primeiro caractere do grupo será sempre a interrogação (`?`).

Sintaxe:

```
(?<IDENTIFICADOR><CONTEÚDO>)
```

Onde o `IDENTIFICADOR` determina o tipo de operador e `CONTEÚDO` é o trecho da expressão regular que será afetado.

4.4 - Alguns operadores estendidos

Tendo em vista que estamos falando de expressões regulares no **shell GNU**, e que nem todos os recursos PCRE estão implementados, nós vamos nos limitar a alguns operadores estendidos mais comuns que podem ser bastante úteis:

Operador	Descrição
`(?#TEXTO)`	Comentário

`(?:PADRÃO)`	Grupo não capturado
`(?=PADRÃO)`	Asserção positiva à frente de comprimento zero
`(?!PADRÃO)`	Asserção negativa à frente de comprimento zero
`(?<=PADRÃO)`	Asserção positiva anterior de comprimento zero
`(?<!PADRÃO)`	Asserção negativa anterior de comprimento zero
`\K`	Exclui os padrões casados anteriormente

Comentários '(?#TEXTO)'

O agrupamento é ignorado e não participa do casamento:

```
:~$ str='muito quente leite quente'
:~$ grep -Po '(?# Eu sou um comentário)(leite|muito) quente' <<< $str
muito quente
leite quente
```

Grupo não capturado '(?:PADRÃO)'

O subpadrão participa do casamento, mas não recebe um registro numerado (`\1`, `\2`, etc…).

Isso funciona muito bem:

```
:~$ grep -P '(quero)-\1' <<< 'quero-quero'
quero-quero
```

Mas, isso provoca um erro:

```
:~$ grep -P '(?:quero)-\1' <<< 'quero-quero'
grep: reference to non-existent subpattern
```

Asserção positiva à frente '(?=PADRÃO)'

Uma **asserção positiva** é a descrição de um padrão agrupado que deve encontrar, obrigatoriamente, um casamento na string, mas que não será tratado como parte do resultado: daí ser definido como *de comprimento zero*. No caso do operador de asserção positiva **à frente** (*lookahead*, "olhe adiante"), o mecanismo de avaliação da expressão regular verifica se o padrão agrupado adiante possui correspondência para considerar se haverá ou não um casamento com a string:

```
:~$ str=$'salada mista\nsalada de frutas'
:~$ grep -P 'salada(?= mista)' <<< $str
[salada] mista
:~$ grep -Po 'salada(?= mista)' <<< "$str"
salada
```

Isso é útil em situações onde queremos obter apenas uma parte no começo da linha encontrada pelo `grep` sem a necessidade de encadear outras ferramentas numa *pipeline*. No `sed`, isso não faz muito sentido, já que nós podemos fazer buscas e substituições de textos.

> ***Importante:*** *os operadores de comprimento zero não registram grupos!*

```
:~$ grep -P 'quero-(?=quero)' <<< 'quero-quero'
[quero-]quero
:~$ grep -P 'quero-(?=quero) \1' <<< 'quero-quero'
grep: reference to non-existent subpattern
```

Asserção negativa à frente '(?!PADRÃO)'

Ao contrário da asserção positiva, uma asserção negativa determina um padrão que **não pode ser casado**:

```
:~$ str=$'salada mista\nsalada de frutas'
:~$ grep -P 'salada(?! mista)' <<< "$str"
```

```
[salada] de frutas
```

Asserção positiva anterior '(?<=PADRÃO)'

Funciona como a asserção positiva à frente, mas condicionando os padrões seguintes ao casamento do padrão agrupado anteriormente (***lookbehind**, "olhe para trás"):

```
:~$ str=$'muito quente\nleite quente'
:~$ grep -P '(?<=leite )quente' <<< "$str"
leite [quente]
```

Asserção negativa anterior '(?<!PADRÃO)'

Condiciona os padrões seguintes ao **não casamento** de um padrão que tenha sido agrupado anteriormente:

```
:~$ str=$'muito quente\nleite quente'
:~$ grep -P '(?<!leite )quente' <<< "$str"
muito [quente]
```

O operador '\K'

Introduzido na PCRE7.2, o operador `\K` também estabelece uma asserção positiva anterior (*lookbehind*), mas não está limitada a um padrão agrupado: ou seja, todos os padrões que vierem antes dele participarão do mecanismo de casamento, mas serão descartados no resultado da avaliação da expressão regular.

Por exemplo, se quiséssemos apenas o último campo da linha de `/etc/passwd` iniciada por `blau`, seria um problema descartar toda a parte anterior do casamento:

```
:~$ grep -Po '^blau.*:/bin/bash' /etc/passwd
blau:x:1000:1000:Blau Araujo,,,:/home/blau:/bin/bash
```

Com o operador `\K`, a solução fica bem mais simples:

```
~ $ grep -Po '^blau.*:\K/bin/bash' /etc/passwd
/bin/bash
```

5 - Qual é a sua intenção?

Sem uma aplicação, a descrição de padrões, seja com os curingas da formação de nomes de arquivos ou com expressões regulares, é apenas isso: a descrição de um padrão de texto. Sendo assim, podemos dizer que a expressão abaixo está correta para descrever, por exemplo, *três caracteres alfabéticos minúsculos quaisquer em sequência*:

```
# Curingas/regex
[a-z][a-z][a-z]
```

No entanto, se formos utilizá-la em algum contexto, elas produzirão resultados bem diferentes:

```
# Expande arquivos no diretório corrente cujos nomes possuam
# exatamente 3 caracteres alfabéticos minúsculos quaisquer.

echo [a-z][a-z][a-z]

# Afirma que a string da esquerda tem exatamente três caracteres
# alfabéticos minúsculos quaisquer.

[[ banana == [a-z][a-z][a-z] ]] # (resulta em erro)

# Afirma que a string da esquerda contém uma sequência de três
# caracteres alfabéticos minúsculos quaisquer.

[[ banana =~ [a-z][a-z][a-z] ]] # (resulta em sucesso)
```

Se os resultados da aplicação das expressões nesses três contextos são os esperados ou não, depende da sua **intenção**.

O que é uma descrição correta

Na prática, para ser considerada *"correta"*, não basta que a expressão descreva os caracteres que esperamos encontrar no texto: é preciso que ela reflita precisamente o que pretendemos obter e se adeque ao funcionamento dos comandos e programas onde ela será utilizada.

Vamos escrever a descrição anterior de outro modo para demonstrarmos o que estamos dizendo com o `egrep` (é o mesmo que `grep -E`):

```
:~$ egrep '[a-z]{3}' <<< abcde
[abc]de
```

Observe que o `grep` encontrou um casamento nos três primeiros caracteres da string (aqui, representado entre colchetes). Mas, se pensarmos que, para o `grep`, a string é vista como uma **linha de texto**, o que realmente aconteceu foi a *localização de uma linha que continha o padrão descrito na regex*.

Então, se a nossa **intenção** era encontrar linhas que contivessem o padrão descrito na regex, nós conseguimos! Porém, a nossa intenção também poderia ser imprimir apenas as sequências de três caracteres minúsculos que o `grep` encontrasse nas linhas, o que seria possível com a opção `-o`:

```
:~$ egrep -o '[a-z]{3}' <<< abcde
abc
```

Excelente! Mas se, em vez disso, a nossa intenção fosse encontrar apenas as linhas que casassem **exatamente** com o padrão, ou seja, linhas compostas apenas por sequências de três caracteres alfabéticos minúsculos, nós poderíamos recorrer à opção `-w`:

```
:~$ egrep -w '[a-z]{3}' <<< abcde
:~$ # sem resultados = estado de saída: erro
:~$ egrep -w '[a-z]{3}' <<< abc
abc # casou = estado de saída: sucesso
```

Portanto, alterando apenas as opções do utilitário `grep`, nós conseguimos expressar as nossas intenções sem alterar a descrição feita na regex: ou seja, para essas aplicações e utilizando **as capacidades de um programa específico**, a descrição se mostrou correta.

Se a descrição está correta, vamos tentar solucionar os mesmos problemas com o comando composto `[[` e o operador `=~`. Primeiro, vamos verificar se o padrão está contido na string (aqui é string mesmo, não uma linha do texto):

```
:~$ [[ abcde =~ [a-z]{3} ]]; echo $?
0
```

Sucesso! Mas, como eu sei disso?

O parâmetro especial `?` expande o estado de saída do último comando executado, que é expresso pelo inteiro `0`, em caso de sucesso, ou qualquer valor diferente de zero no caso de erro. Como expandiu `0`, eu sei que a minha afirmação foi avaliada como verdadeira e o comando composto `[[` saiu com sucesso.

Agora, vamos imprimir o que foi casado, o que pode ser feito com a expansão do elemento `0` do vetor `BASH_REMATCH`:

```
:~$ [[ abcde =~ [a-z]{3} ]]; echo ${BASH_REMATCH[0]}
abc
```

Quando utilizamos o comando composto `[[` com o operador `=~`, o Bash definie o elemento de índice `0` do vetor `BASH_REMATCH` com a parte da string que casar com a regex. Se existissem agrupamentos na expressão, os elementos seguintes (a partir do índice `1`) seriam definidos segundo a ordem dos grupos, cumprindo o papel dos *retrovisores* `\1`, `\2`, etc.

Muito bem, utilizando um comando interno do Bash, nós conseguimos reproduzir os dois primeiros resultados que obtivemos com o `grep`. Só falta encontrar uma forma de conseguir o terceiro e último resultado: a descrição do padrão deve casar **exatamente** com a string… Só que isso não é possível alterando apenas recursos do comando `[[`: nós teremos que encontrar outra descrição que exprese a nossa intenção:

```
:~$ [[ abcde =~ ^[a-z]{3}$ ]]; echo $?
1
:~$ [[ abc =~ ^[a-z]{3}$ ]]; echo $?
0
```

Neste caso, nós fomos obrigados a utilizar as âncoras `^` e `$` para indicar, respectivamente o início e o fim da string. Isso quer dizer que, neste contexto, com as limitações dos recursos do comando utilizado, a descrição original não era a correta para expressar a nossa intenção.

O mesmo tipo de cuidado deve ser observado em outras situações…

Localização do padrão na string

Tipicamente, quando a intenção é confirmar ou localizar um padrão na string, nós só utilizamos âncoras quando queremos especificar onde, na string, o padrão deve casar. Se a descrição for feita com curingas, o asterisco pode ser usado para indicar onde o casamento **não precisa** acontecer:

```
:~$ [[ abcde =~ ^[a-z]{3} ]]; echo ${BASH_REMATCH[0]}
abc
:~$ [[ abcde =~ [a-z]{3}$ ]]; echo ${BASH_REMATCH[0]}
cde
:~$ [[ cdefg == cde* ]]; echo $?
0
:~$ [[ cdefg == *cde ]]; echo $?
1
:~$ [[ cdefg == *cde* ]]; echo $?
0
```

Casamento integral

O casamento integral pode ser necessário em várias situações, como:

- Validação de entradas de dados
- Substituições de linhas inteiras (`sed`)
- Expansão de nomes de arquivos

No caso das expansões, as descrições (apenas com curingas) sempre expressarão o que deve casar integralmente com os nomes dos arquivos, mas não devemos confundir o mecanismo da expansão com outros contextos onde os curingas são utilizados para descrever padrões, como no comando `[[` ou em cláusulas do `case`.

Nos casos de uso das expressões regulares, o utilitário `sed` é um bom exemplo de como a especificação dos limites do padrão pode ser relevante:

```
:~$ sed -E 's/([a-z]{4})/\1/' <<< 'minha casa branca'
minha casa branca
:~$ sed -E 's/.* ([a-z]{4}) .*/\1/' <<< 'minha casa branca'
casa
```

O cuidado com a especificação dos limites pode ser mais relevante ainda quando temos que validar algum tipo de dado recebido:

```
:~$ [[ 1234 =~ [0-9]{4} ]]; echo $?
0
:~$ [[ 12345 =~ [0-9]{4} ]]; echo $?
0 <--- Um falso sucesso!
:~$ [[ 12345 =~ ^[0-9]{4}$ ]]; echo $?
1 <--- Agora sim!
```

Exercícios

1 - Sobre padrões de nomes de arquivos

Considere o diretório abaixo nas suas respostas:

```
:~$ ls -a
.                 compras.txt     'meus pensamentos.txt'   punk.mp3
..                .config          mpb.mp3                 rock.mp3
aniversário.png   currículo.odt    natal.png               .segredos
biografia.odt     ficante.png      notas.txt               viagem.png
bossa.mp3         jazz.mp3         pagode.mp3
```

1 - Quais nomes de arquivos casam com o padrão abaixo?

```
???[ak]*[3g]
```

2 - Como obter todos os arquivos `.png`, exceto `ficante.png`?

3 - Considerando a saída abaixo, como evitar a expansão de `.` e `..`?

```
:~$ echo .*
. .. .config .segredos
```

4 - Como expandir apenas `currículo.odt` e `aniversário.png`?

5 - Escreva um padrão que case apenas com nomes de arquivos que contenham espaços.

6 - Com apenas uma expressão, expanda somente os nomes de arquivos abaixo:

```
compras.txt .config currículo.odt notas.txt rock.mp3
```

Responda as próximas questões usando as palavras abaixo:

```
cana banana ponte 'Belo Horizonte' bandana entulho bagulho
bacana contente patente colante savana
```

Nota: você deverá criar um script para testar todas as palavras da lista.

7 - Use o operador `==` do *teste do bash* para casar apenas com:

```
ponte 'Belo Horizonte' entulho
```

8 - Quais palavras casarão com o padrão `*(ba)?(nda|ca)+(na)`?

9 - Com uma expressão, como casar apenas com as palavras abaixo:

```
colante patente ponte entulho
```

10 - Crie um script contendo um `case` capaz de separar as palavas da lista em grupos de padrões: todas as palavras têm que casar com algum grupo.

2 - Busca e substituição em um arquivo HTML

Eu tenho o seguinte script:

```
#!/bin/bash

url='https://www.gnu.org/software/bash/manual/bash.html'
src='bash-index.txt'

declare -A toc

while read line; do
   toc[${line% *}]="${line##* }"
done < $src

sel=$(fzf <<< $(printf '%s\n' "${!toc[@]}"))

w3m $url${toc[$sel]}
```

Mas, para que ele funcione, eu preciso obter uma lista de tópicos e suas respectivas âncoras do manual oficial do Bash e salvá-la no arquivo `bash-index.txt`.

Complete a linha do `sed`, abaixo, com uma expressão regular que encontre os títulos dos tópicos e suas respectivas âncoras:

```
sed -En 's|REGEX|\2 \1|p'
```

3 - Resolução dos monitores conectados

Observe, abaixo, como eu estou usando o `xrandr`:

```
xrandr --nograb --current
```

Ele me dá a seguinte saída:

```
Screen 0: minimum 320 x 200, current 1920 x 1080, maximum 16384 x
16384
VGA-1 disconnected (normal left inverted right x axis y axis)
HDMI-1 connected primary 1920x1080+0+0 (normal left inverted right x
axis y axis) 160mm x 90mm
  1920x1080     60.00*+  50.00    59.94    30.00    25.00    24.00
29.97    23.98
  1920x1080i    60.00    50.00    59.94
  1280x1024     60.02
  1360x768      60.02
  1280x720      60.00    50.00    59.94
  1024x768      60.00
  800x600       60.32
  720x576       50.00
  720x576i      50.00
  720x480       60.00    59.94
  640x480       60.00    59.94
  720x400       70.08
DP-1 disconnected (normal left inverted right x axis y axis)
```

Repare que, ao lado de cada dispositivo de vídeo, nós temos as palavras `connected` e `disconnected`. Com base nisso, complete a *pipeline* abaixo para que eu tenha na saída o nome de cada dispositivo conectado seguido de sua resolução, por exemplo:

```
:~$ echo $my_res
VGA-1:1920x1080 HDMI-1:1920x1080
```

A pipeline:

```
my_res=$(echo $(xrandr --nograb --current | sed -nE
's/REGEX/SUBST/p'))
```

4 - Tema GTK e de ícones

Considerando as minhas configurações no arquivo abaixo:

```
:~$ cat ~/.config/gtk-3.0/settings.ini
[Settings]
gtk-theme-name=Arc-Dark
gtk-icon-theme-name=debxp-paper-folders
gtk-font-name=Open Sans 10
gtk-cursor-theme-name=breeze_cursors
gtk-cursor-theme-size=0
gtk-toolbar-style=GTK_TOOLBAR_BOTH
gtk-toolbar-icon-size=GTK_ICON_SIZE_LARGE_TOOLBAR
gtk-button-images=1
gtk-menu-images=1
gtk-enable-event-sounds=1
gtk-enable-input-feedback-sounds=1
gtk-xft-antialias=1
gtk-xft-hinting=1
gtk-xft-hintstyle=hintfull
gtk-xft-rgba=rgb
gtk-decoration-layout=appmenu
```

Complete as expressões regulares abaixo para obter o tema GTK e o tema de ícones nas variáveis abaixo:

```
# Meu arquivo de configurações...
gtk_settings=$HOME/.config/gtk-3.0/settings.ini

# As variáveis...
my_theme=$(grep -Po 'REGEX' $gtk_settings)
my_icons=$(grep -Po 'REGEX' $gtk_settings)
```

Resultado esperado:

```
:~$ echo $my_theme
Arc-Dark
:~$ echo $my_icons
debxp-paper-folders
```

5 - Nome da distribuição

Considere a saída do comando abaixo:

```
:~$ grep PRETTY /etc/os-release
# PRETTY_NAME="Debian GNU/Linux bookworm/sid"
PRETTY_NAME="Debian GNU/Linux-libre (sid)"
```

Complete a REGEX abaixo de modo a obter a seguinte saída:

```
:~$ sed -nE 's/REGEX/SUBST/p' /etc/os-release
Debian GNU/Linux-libre/sid
```

6 - Algumas validações simples

Construa expressões regulares para validar os seguintes dados abaixo.

Endereço de e-mail:

- Letras, números, pontos e traços
- Seguidos de `@`
- Seguido de apenas letras
- Seguidas de ponto
- Seguido de apenas 3 letras
- Opcionalmente, seguidas de um ponto e até duas letras.

Endereço Ipv4:

`0-255.0-255.0-255.0-255`

Não pode casar com números com zeros à esquerda!

Data e hora:

Casar com o formato de saída do comando `date '+%F %R'`.

7 - HTML para markdown

Considerando o trecho em HTML abaixo, converta as marcações para markdown.

```
<p>Maria <strong>tinha</strong> um carneirinho</p>
<p>Sua lã era <em>branca <strong>como</strong> a neve</em>.</p>
<p>Aonde <code>quer que</code>  menina fosse</p>
<p>O carneirinho ia atrás.</p>
```

Resultado esperado:

```
Maria **tinha** um carneirinho
Sua lã era *branca **como** a neve*.
Aonde `quer que`  menina fosse
O carneirinho ia atrás.
```

8 - Markdown para HTML

Converta o resultado do exercício anterior novamente para HTML.

Resultado esperado:

```
<p>Maria <strong>tinha</strong> um carneirinho</p>
<p>Sua lã era <em>branca <strong>como</strong> a neve</em>.</p>
<p>Aonde <code>quer que</code>  menina fosse</p>
<p>O carneirinho ia atrás.</p>
```