PCS3616 - Laboratório 2 - Introdução ao UNIX 2

Ferramentas básicas

Programação com vim, gcc, gdb e make

Introdução

Nessa aula, você vai aprender para que servem e como usar algumas ferramentas que irá encontrar em qualquer projeto de software de porte razoável. Quanto mais complexo o projeto (em termos de quantidade de código e de componentes, número de pessoas envolvidas, dependências de bibliotecas externas, etc), mais essas ferramentas se tornam importantes para auxiliá-lo a se manter produtivo e colocar o computador para trabalhar a seu favor.

O conjunto das ferramentas usadas em um projeto de software é normalmente conhecido como toolchain. Dependendo da complexidade do projeto, a sua *toolchain* irá conter um número maior ou menor de ferramentas, e isso vale para o desenvolvimento de qualquer sistema e em qualquer linguagem de programação. Para deixar a discussão mais concreta, vamos falar sobre desenvolvimento em C, que é uma das linguagens mais usadas quando se trata de software básico.

Para escrever e executar um programa em C, o mínimo que você precisa é de:

- Um **editor de texto**, para escrever o código-fonte do programa;
- Um **compilador**, para converter o código-fonte em um arquivo executável escrito em linguagem de máquina (código binário). O executável resultante do processo de compilação será interpretado e executado pelo sistema operacional e pelo hardware.

A princípio, é só isso! Com essas duas ferramentas, você já pode escrever qualquer programa imaginável e executá-lo, mas existem muitas outras ferramentas que vão tornar a sua vida muito mais fácil se o seu programa for minimamente complexo.

Por exemplo, você roda o programa e descobre que ele não está se comportando *exatamente* como esperado... Às vezes ele produz o resultado certo, às vezes não. Relendo o código, você não encontra nada de errado. Como descobrir o que está acontecendo? Agora é um bom momento para introduzir mais uma ferramenta:

- Um **debugger**, que permite que você execute o seu programa passo-a-passo, inspecionando como o estado do programa vai sendo alterado durante a execução—por exemplo, mudanças nos valores de variáveis. Com um debugger, você pode acompanhar a execução do programa desde o início ou interromper a execução quando ele chegar em uma determinada linha de código.

Uma outra etapa do processo de desenvolvimento é que comumente automatizada é a etapa de *build*. Por exemplo, digamos que toda vez que você faz alguma alteração no software, precisa realizar os seguintes passos:

- Recompilar o projeto (para projetos muito grandes e dependendo da capacidade do seu computador, a compilação de todos os arquivos pode demorar de minutos a algumas horas);
- Executar testes automatizados com o executável—por exemplo, para garantir que a correção de um bug não introduziu novos bugs em outros componentes;
- Mover o executável para o diretório onde o seu sistema operacional espera encontrar executáveis (que normalmente é qualquer diretório no seu $PATH).

Os passos acima podem ser automatizados de diversas formas. Uma delas é escrever um *shell script*, simplesmente. A outra, muito comum, é usar uma **ferramenta de build**, que permite que você declare receitas para obter determinados artefatos a partir de outros—por exemplo, como obter um arquivo executável a partir dos arquivos-fonte do seu projeto.

Para cada uma das ferramentas citadas acima, existem diversas opções de programas, acessíveis, inclusive, pela linha de comando. A tabela a seguir lista alguns exemplos.

Ferramenta	**Programas**
Editor de texto	nano, vi/vim, emacs, gedit
Compilador	gcc, clang, icc, javac (Java)
Debugger	gdb, lldb
Ferramenta de build	make, rake (Ruby), ant (Java)

Note que o compilador e o debugger são ferramentas que dependem da linguagem de programação usada para desenvolvimento, enquanto o editor de texto e a ferramenta de build normalmente são independentes de uma linguagem.

No restante do tutorial, vamos falar sobre um programa para cada ferramenta: **vim**, **gcc**, **gdb** e **make**.

Editor de texto: vim

O vim é um editor de texto usado diretamente pela linha de comando (apesar de também poder ser usado com uma interface gráfica). Para editar um arquivo de texto com o vim, basta digitar:

vim <caminho_para_o_arquivo>

Por exemplo:

vim ~/.bashrc

O vim funciona de uma maneira um pouco diferente da que você provavelmente está acostumado. Ele possui dois modos de operação distintos: modo de comando (*command mode*) e modo de edição (*insert mode*).

O modo de comando, que é o modo em que o vim se encontra quando é iniciado, permite que você navegue pelo texto e faça alterações no arquivo usando... *wait for it*... comandos.

Por exemplo, se você abrir um arquivo no vim, levar o cursor até uma determinada letra e apertar a tecla x, o caractere em que o cursor está posicionado será excluído, ao invés da letra "x" ser adicionada ao texto. Ou seja: **no modo de comando, as teclas do seu teclado servem para executar comandos, ao invés de inserir caracteres no texto.**

O modo em que você usa o teclado para realmente digitar caracteres é o modo de edição. Para entrar no modo de edição, o comando é a tecla i ou a tecla insert. Se você estiver no modo de edição, o texto -- INSERT -- será mostrado no canto inferior esquerdo da tela.

Sobre o modo de edição não há muito o que falar porque, neste modo, o vim se comporta como qualquer editor comum.

Sobre o modo de comando, o que você precisa saber são os comandos. Vamos mostrar alguns aqui, mas existem muitos outros. A tabela abaixo lista alguns dos mais comuns:

Comando	**Descrição**
:q	Sai do editor (*quit*)
:w	Salva o arquivo (*write*)
:wq	Salva o arquivo e sai (*write and quit*)
:q!	Sai do editor (descartando alterações não salvas)
i	Entra no modo de edição (*insert mode*)
Esc	Entra no modo de comando (*command mode*)
j, k, l, h	Move o cursor para baixo, cima, direita e esquerda, respectivamente
gg	Move o cursor para o início do arquivo

Shift + g	Move o cursor para o final do arquivo
$	Move o cursor para o final da linha
0	Move o cursor para o início da linha
:N	Move o cursor para a linha N
dd	Apaga a linha atual

Compilador: gcc

O gcc é um compilador: um programa que permite converter um ou mais arquivos-fonte escritos em C em um arquivo executável. Na verdade, falando mais rigorosamente, o gcc é um *compiler driver*, pois ele executa outros passos além da compilação, mas não vamos entrar nesses detalhes ainda.

O tópico de compilação de um programa (independente das linguagens de entrada e de saída) é uma das áreas mais complexas e interessantes da Computação. Um dos motivos para a complexidade é que existem muitas decisões que precisam ser tomadas ao longo do processo de compilação.

Por exemplo: qual a plataforma em que o programa será executado? Muitas vezes, você compila um programa em um computador, mas irá executá-lo em outro. Você gostaria de otimizar o código gerado? Se sim, gostaria de otimizar de acordo com qual parâmetro de desempenho: tamanho do executável, uso de memória (*memory footprint*), tempo de execução, ...? Onde o compilador deve procurar para encontrar bibliotecas externas que serão usadas pelo seu programa?

Dependendo da sua necessidade, o uso do gcc pode ficar bastante complexo. No entanto, para casos de uso simples, a linha de comando é igualmente simples:

gcc -o <arquivo_de_saida> <arquivo_de_entrada>

Por exemplo:

gcc -o foo foo.c

Depois de executar esse comando, se não houver nenhum erro, você poderá executar o programa pelo terminal:

./foo

Exercício 1.1: Utilize o template abaixo para criar o arquivo fatorial.c. Usando o vim, edite o arquivo de tal forma que o programa funcione corretamente (veja os comentários no código). Em seguida, compile e execute o programa usando o gcc.

Template:

```c
#include <stdio.h>

int fatorial(int n);

/**
 * Ponto de entrada do programa (entry point).
 */
int main(int argc, char** argv) {
  int n;
  scanf("%d", &n);

  printf("%d\n", fatorial(n));

  return 0;
}

/**
 * Calcula o fatorial do número recebido e retorna o resultado.
 */
int fatorial(int n) {
  // Por enquanto, essa função não faz nada de muito útil... Ela apenas
  // retorna o número recebido.
  // Altere esta implementação para retornar o resultado correto.
  return n;
}
```

Exemplo de execução:

```
$ ./fatorial
4
24
```

Entrega: o arquivo fatorial.c

Atenção: o algoritmo será corrigido automaticamente. Para que ele possa funcionar corretamente, em todo o corpo do código, receba apenas um único número de entrada, o valor a ser calculado seu fatorial, e imprima uma única saída, o fatorial correspondente. Caso o código apresente falha de execução ou não passe em todos os testes de avaliação, você poderá editá-lo e submetê-lo novamente.

Debugger: gdb

O gbd é um **debugger**: um programa que permite acompanhar a execução de um programa e também interferir nela. O gdb funciona com muitas linguagens, mas as mais comuns são C e C++.

O uso do gdb será demonstrado a partir do exemplo abaixo, que mostra como descobrir a causa de um erro de *segmentation fault* em um programa. Esse exemplo é uma tradução/adaptação do original disponível aqui.

Usando o gdb para resolver um problema de *segfault*

Neste exemplo, você irá aprender a usar o gdb para descobrir por que o programa abaixo, quando executado, causa um erro de *segmentation fault*.

O programa deveria ler uma linha de texto do usuário e imprimi-la. No entanto, veremos que, do jeito como está, o resultado não é exatamente esse...

Versão inicial (com bug) do programa:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  printf("Digite um texto qualquer e pressione Enter ao final:\n");

  char *buf;
  buf = malloc(1<<31);
  fgets(buf, 1024, stdin);

  printf("Texto digitado foi:\n");
  printf("%s\n", buf);

  return 1;
}
```

Crie um arquivo chamado segfault-example.c, com o conteúdo acima. Compile o arquivo usando o gcc e o execute, para verificarmos se realmente existe um problema:

```
gcc -o segfault-example segfault-example.c
./segfault-example
```

Você deve ter observado que, realmente, o programa termina com erro antes de imprimir a linha digitada pelo usuário. Vamos usar o gdb, então, para descobrir o motivo.

O primeiro passo é recompilar o arquivo usando a flag -g, que faz com que o gcc compile o programa de uma forma ligeiramente diferente, incluindo símbolos que serão usados pelo gdb.

```
gcc -g -o segfault-example segfault-example.c
```

Agora, execute o programa novamente, dessa vez pelo gdb:

```
$ gdb segfault-example
[várias linhas na saída, omitidas]
Reading symbols from segfault-example...done.
(gdb)
```

Nesse ponto, o gdb está pronto e aguardando instruções. Para começar, vamos simplesmente executar o programa usando o comando run e ver o que acontece:

```
(gdb) run
Starting program:
/home/deborasetton/Documents/Mestrado/Monitoria/PCS3616-Systems-Programmi
ng/aula2/segfault-example
Digite um texto qualquer e pressione Enter ao final:
QUALQUER COISA AQUI

Program received signal SIGSEGV, Segmentation fault.
__GI__IO_getline_info (fp=fp@entry=0x7ffff7dd3980 <_IO_2_1_stdin_>,
buf=buf@entry=0x0, n=1022, delim=delim@entry=10,
extract_delim=extract_delim@entry=1, eof=eof@entry=0x0) at iogetline.c:86
86      iogetline.c: No such file or directory.
(gdb)
```

O gdb executa o programa até onde consegue e para novamente, informando que recebeu o sinal SIGSEGV do sistema operacional. Isso significa que o programa tentou acessar uma região de memória inválida.

Vamos usar o comando backtrace para descobrir onde exatamente o programa travou:

```
(gdb) backtrace
#0  __GI__IO_getline_info (fp=fp@entry=0x7ffff7dd3980 <_IO_2_1_stdin_>,
buf=buf@entry=0x0, n=1022, delim=delim@entry=10,
extract_delim=extract_delim@entry=1, eof=eof@entry=0x0) at iogetline.c:86
#1  0x00007ffff7a7f188 in __GI__IO_getline (fp=fp@entry=0x7ffff7dd3980
<_IO_2_1_stdin_>, buf=buf@entry=0x0, n=<optimized out>,
delim=delim@entry=10, extract_delim=extract_delim@entry=1) at
iogetline.c:38
#2  0x00007ffff7a7dfc4 in _IO_fgets (buf=0x0, n=<optimized out>,
```

```
fp=0x7ffff7dd3980 <_IO_2_1_stdin_>) at iofgets.c:56
#3  0x0000000000400647 in main (argc=1, argv=0x7fffffffd5c8) at
segfault-example.c:10
(gdb)
```

Repare que a saída do comando faz referência a alguns arquivos que o programa usa, mas que não fomos nós que escrevemos, como iogetline.c e iofgets.c.

Como estamos interessados no nosso próprio código, vamos usar o comando frame para ir até o frame **3**, que é o frame que fala sobre o nosso arquivo, segfault-example.c:

```
(gdb) frame 3
#3  0x0000000000400647 in main (argc=1, argv=0x7fffffffd5c8) at
segfault-example.c:10
10        fgets(buf, 1024, stdin);
(gdb)
```

Ok, então o programa travou na chamada à função fgets. De maneira geral, sempre podemos assumir que funções da biblioteca padrão, como esta, estão funcionando -- se este não for o caso, o problema é muito maior.

Portanto, o problema deve estar em um dos 3 argumentos que passamos para a função. Talvez você não saiba, mas stdin é uma variável global que é criada pela biblioteca stdio, então este argumento podemos assumir que está ok. Resta o argumento buf.

Vamos usar o comando print para inspecionar o valor desta variável:

```
(gdb) print buf
$1 = 0x0
(gdb)
```

O valor da variável é 0x0, que é o ponteiro nulo. Isso não é o que queremos -- buf deveria apontar para uma área de memória que foi alocada na chamada ao malloc (veja o código).

Portanto, vamos ter que descobrir o que aconteceu aqui. Mas antes, podemos encerrar a instância atual do programa (que já nos deu informações suficientes e não tem mais o que executar) usando o comando kill:

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

Após este comando, estamos novamente no início do gdb. Desta vez, vamos colocar um breakpoint na linha do código que chama o malloc:

```
(gdb) break segfault-example.c:9
Breakpoint 1 at 0x40062f: file segfault-example.c, line 9.
(gdb)
```

Agora, vamos rodar o programa novamente:

```
(gdb) run
Starting program:
/home/deborasetton/Documents/Mestrado/Monitoria/PCS3616-Systems-Programmi
ng/aula2/segfault-example
Digite um texto qualquer e pressione Enter ao final:

Breakpoint 1, main (argc=1, argv=0x7fffffffd5c8) at segfault-example.c:9
9         buf = malloc(1<<31);
(gdb)
```

Primeiro, vamos ver qual é o valor de buf antes da chamada ao malloc, usando o comando print. Uma vez que essa variável ainda não foi inicializada, esperamos que o valor seja inválido, e realmente é:

```
(gdb) print buf
$1 = 0x0
(gdb)
```

Agora, vamos usar o comando next para executar apenas esta linha de código e parar novamente, para podermos ver o que aconteceu com a variável:

```
(gdb) next
10        fgets(buf, 1024, stdin);
(gdb) print buf
$2 = 0x0
(gdb)
```

Após a chamada, verificamos que buf continua inválido, apontando para NULL. Por quê? Se você consultar a documentação do malloc, descobrirá que essa função retorna NULL quando não consegue alocar a quantidade de memória solicitada. Portanto, a chamada ao malloc feita pelo nosso programa deve ter falhado. Vamos olhar esssa chamada novamente:

```
buf = malloc(1<<31);
```

Bom... O valor da expressão 1 << 31 (o inteiro 1 deslocado 31 bits à esquerda) é 429497295 ou 4GB. Poucos sistemas operacionais alocariam esta quantidade de memória para um único programa, a não ser com configurações especiais, então é claro que o malloc

falhou. Além disso, nós só estamos lendo 1024 bytes com o fgets, então para que alocar tanta memória?

Mude o valor 1<<31 no código-fonte para 1024, compile e execute o programa novamente:

```
$ gcc -o segfault-example segfault-example.c
$ ./segfault-example
Digite um texto qualquer e pressione Enter ao final:
QUALQUER COISA AQUI
Texto digitado foi:
QUALQUER COISA AQUI
```

Problema resolvido! \o/

E agora você sabe como depurar segfaults usando o gdb, o que é extremamente útil. Finalmente, este exemplo também ilustra um outro ponto muito importante: **sempre verifique o valor retornado pelo malloc!**

Debugger: make

A última ferramenta de que vamos falar aqui é o make, que permite a automatização de muitas operações, e é comumente utilizado para gerenciar o processo de build de um software.

O make é basicamente uma ferramenta que executa comandos do shell condicionalmente (normalmente, dependendo do estado de um ou mais arquivos). Um caso de uso muito comum para o make é o seguinte: imagine um software composto por centenas de arquivos em C. Compilar todos esses arquivos e construir os executáveis necessários leva, digamos, 40 minutos. Você estava corrigindo um bug no sistema e, para corrigir o bug, precisou alterar apenas 3 arquivos do projeto; por volta de 10 linhas foram modificadas ao todo.

Agora, você precisa obter novos executáveis, compilados a partir do código modificado. Não é possível recompilar apenas os 3 arquivos que foram alterados, porque existem outros arquivos que dependem destes 3, então eles precisarão ser recompilados também. E, então, os arquivos que dependem *desses outros* também precisarão ser recompilados... O que você acha de montar essa lista de quais arquivos precisarão ser recompilados manualmente, olhando arquivo por arquivo? Ou será que é melhor compilar tudo do zero e esperar os 40 minutos de uma vez?

Nenhuma dessas opções é ideal, mas a boa notícia é que o make ajuda a resolver exatamente esse tipo de problema: se você disser quais arquivos dependem de quais (e dessa vez é suficiente informar apenas o primeiro nível de dependência), o make utiliza a data de modificação dos arquivos para descobrir se um determinado executável precisa ser recompilado ou não, com base na data de modificação de suas dependências.

O arquivo que informa o make sobre as dependências de cada arquivo, contém instruções sobre como (re-)construir executáveis, executar testes, etc em um projeto é o Makefile.

Makefile

Um Makefile é um arquivo que contém **regras** (*rules*). Cada regra define como construir um ou mais ***targets*** a partir de dois itens: uma lista de dependências (*prerequisites*, que pode ser vazia) e uma **receita** (*recipe*), que são apenas comandos que serão executados em um shell.

A sintaxe básica de uma regra de um Makefile é:

```
# Sintaxe de uma regra.
# Linhas que começam com '#', como estas, são comentários.

targets : prerequisites
    recipe_line_1
    recipe_line_2
    ...
    recipe_line_n
```

Uma observação importante é que todas as linhas de uma receita devem começar com o caractere Tab (\t). Por isso, atenção para usar Tabs e não espaços. O seu editor de texto pode ser configurado (se já não estiver) para mostrar espaços em branco no texto, distinguindo tabs de espaços.

Exemplo de Makefile

Saindo um pouco do contexto de programação, o arquivo abaixo é um Makefile que define como construir um arquivo groceries.txt a partir de outros dois arquivos, fruits.txt e vegetables.txt. Quem nunca quis controlar a lista do supermercado usando Makefiles, certo?!

```
SHELL=/bin/bash

# Este Makefile contém 4 regras, descritas abaixo.

# Esta primeira regra contém apenas um target: o arquivo groceries.txt.
# Para que o make consiga construir o arquivo, outros dois arquivos estão
# listados como dependências: um com a lista de frutas e outro com a
# lista de legumes.
# Se as dependências forem satisfeitas, o make executa as linhas de
# receita, que apenas imprime o conteúdo dos dois arquivos em um arquivo
# final, acrescentando linhas de cabeçalho.
# Importante: cada linha é executada em uma nova instância nova de um
```

```
# shell (o que significa, por exemplo, que variáveis definidas em uma
# linha não estarão disponíveis nas linhas seguintes).
#
#------------------------------------------------------------------------
groceries.txt : fruits.txt vegetables.txt
    echo -e "Fruits:\n" > groceries.txt
    cat fruits.txt >> groceries.txt
    echo -e "\nVegetables:\n" >> groceries.txt
    cat vegetables.txt >> groceries.txt

# Esta regra contém dois targets. Nenhum deles tem dependências, e a
# receita apenas imprime uma mensagem dizendo para o usuário criar o
# arquivo correspondente.
#
#------------------------------------------------------------------------
fruits.txt vegetables.txt:
    echo "Please create the file $@"

# Esta é uma regra auxiliar. O target `clean` não é um arquivo que será
# criado; a regra é basicamente um atalho para executar uma linha de
# comando. Nesse caso, a receita apaga o arquivo groceries.txt.
#
#------------------------------------------------------------------------
clean:
    rm -f groceries.txt

# Esta é uma outra regra auxiliar, que imprime o conteúdo do arquivo de
# saída (o que significa que faz sentido listá-lo como dependência).
#
#------------------------------------------------------------------------
print : groceries.txt
    cat groceries.txt
```

Como usar o make na linha de comando

O uso básico do make é:

```
make <target>
```

Com este comando, o make irá procurar um Makefile (que é apenas um arquivo com este nome) no mesmo diretório em que o comando foi executado (a opção -f pode ser usada caso o Makefile esteja em outro diretório).

Em seguida, irá analisar o Makefile para encontrar uma regra que ensine como construir o *target* passado como parâmetro. Se o *target* tiver dependências pendentes, elas serão construídas primeiro, e assim por diante.

Por exemplo, para construir o arquivo groceries.txt usando o Makefile acima:

```
make groceries.txt
```

Exercício 1.2: criar um Makefile (usando o vim) com o conteúdo do exemplo acima. Em seguida, criar um shell script chamado makefile-test.sh que chama, um por um, todos os targets possíveis desse Makefile (dica: são 5).

Entrega: Um zip (makefile-test.zip) contendo o shell script (makefile-test.sh)

Exercício 1.3: criar um Makefile (usando o vim) para o programa do fatorial. O Makefile deve definir as seguintes regras:

1. Compilar o arquivo fatorial.c (com o método main()) usando o gcc. O nome do target (e do executável produzido) deve ser fatorial.
2. Executar o arquivo fatorial, caso ele exista. Esta regra não deve compilar o arquivo, apenas chamar o executável. Nome do target: run.
3. Executar o arquivo no gdb. Na receita, você deve compilar o arquivo com a flag necessária para usar o gdb e, em seguida, chamar o executável. Nome do target: run_gdb.
4. Remover o executável. Nome do target: clean.

Entrega: Um arquivo zip com nome lab2-atv3-XXXXXX.zip e com os seguintes arquivos:

```
lab2-atv3-XXXXXX.zip/
├── fatorial.c
└── Makefile
```

Expressões regulares

Introdução

O conceito de **expressão regular** é um dos conceitos teóricos mais importantes de Computação, e também é muito usado na prática, em diversos tipos de atividades, como:

- Encontrar um arquivo no seu computador cujo texto contenha um determinado padrão. Por exemplo, encontrar todos os arquivos em um determinado diretório que contenham um número de CPF ou CNPJ.
- Verificar informações fornecidas por usuários que devem obeceder a formatos específicos, como números de telefone, endereços de email, CPF, RG, CEP, etc;
- Verificar se um determinado trecho de código é sintaticamente correto, isso é, se obedece à sintaxe da linguagem de programação em que foi escrito;

- Efetuar substituições de texto em um ou mais arquivos, caso o conteúdo obedeça um determinado padrão. Por exemplo substituir todas as ocorrências de "pcs", "psi" e "pea" em um conjunto de arquivos por "PCS", "PSI" e "PEA", respectivamente;
- Extrair informações em páginas da Web. Por exemplo, capturar os preços de produtos em uma página de busca do Buscapé.

O termo "expressão regular" também é conhecido por **regex** ou **regexp** (contração do termo em Inglês, ***reg**ular **exp**ression*).

Você pode pensar em uma expressão regular como um "padrão" ou uma "forma" que é usada para generalizar e descrever um conjunto de *strings* (sequências de caracteres) sem precisar listar cada string individualmente.

Por exemplo, digamos que você queira encontrar todos os números de CPF em um texto. Como o número de strings que representam CPFs é muito grande para listar um por um, você pode usar uma expressão regular que se "encaixe" em qualquer número de CPF, como este:

<NUM><NUM><NUM><PONTO><NUM><NUM><NUM><PONTO><NUM><NUM><NUM><TRAÇO><NUM><NUM>

Onde: - <NUM> é qualquer dígito entre 0 e 9; - <PONTO> é o caractere de um ponto (.); - <TRAÇO> é o caractere de hífen (-);

Qualquer CPF válido e pontuado obedecerá ao padrão definido acima.

Outro exemplo: um padrão para encontrar números de telefone, sem considerar o DDD. Telefones móveis têm 9 dígitos, enquanto telefones fixos têm 8. Um possível padrão seria:

<NUM_OPCIONAL><NUMx4>-<NUMx4>

Onde: - <NUM_OPCIONAL> é qualquer dígito ou a string vazia; - <NUMx4> é uma sequência de 4 dígitos quaisquer; - O traço (-) representa ele mesmo.

Sintaxe

Como você pode ver, não basta conseguir definir o seu termo de busca como um padrão. Precisamos definir uma sintaxe (um formato) para esses padrões. Vamos usar <TRAÇO> ou - para definir um hífen? Vamos usar <NUM> várias vezes ou poderemos escrever coisas como <NUMx10> quando quisermos especificar sequências do mesmo tipo de caractere?

Quase todas as linguagens e ferramentas que suportam o uso de expressões regulares utilizam a mesma sintaxe e, para poder usar expressões regulares no seu dia-a-dia, você também precisa conhecê-la.

A tabela abaixo é uma referência resumida dessa sintaxe (veja outras referências, mais completas, na seção Recursos Adicionais).

Referência

	Significado	Exemplo	Strings válidas	Strings inválidas
caractere comum	Qualquer caractere que não seja especial corresponde a ele mesmo. Os caracteres especiais são: {}[]()^$.|*+?\ e -, quando aparece dentro de colchetes ([]).	frio	• frio • Hoje está frio. (a regex pode aparecer em qualquer posição da string que está sob análise) • Você prefere frio ou calor?	• Frio (regexes são, por padrão, *case-sensitive*) • fr-io
.	Corresponde a qualquer caractere, com exceção do \n	m.r	• mar • amor • Que horas vamos comer?	• semear
?	O caractere ou grupo precedente é opcional (i.e., pode estar presente ou ausente).	carro?	• carro • carr • carros • carroça	
*	Especifica que o caractere ou grupo que precede o `*` deve aparecer 0 ou mais vezes.	ba*	• b • ba • baa • baaaaaaaaaaa	• a
+	Especifica que o caractere ou grupo que precede o `+` deve aparecer 1 ou mais vezes.	ba+	• ba • baa • baaaaaaaaaaa	• b • a

{n}	Especifica que o caractere ou grupo que precedente deve aparecer exatamente `n` vezes.	ba{3}	• baaa	• b • ba • baaaa
|	Especifica conjunção (`OR`).	ba(nana|rraca)	• banana • barraca	• ba
[abc123]	Colchetes são usados para definir opções. Um grupo entre colchetes corresponde a qualquer um (e apenas um) dos caracteres especificados entre os colchetes.	[2345]	• 2 itens • 3 itens	• 24 itens
\d	Corresponde a qualquer dígito. Equivalente a `[0-9]`.	\d	• 0 itens • 9 itens	• 0 itens • 9 itens
\s	Corresponde a espaços em branco (` `, `\t`, `\n`).	a\s	• a b	• ab
^	Corresponde ao início da string, ou início de uma linha, caso o texto tenha múltiplas linhas.	^v	• você tem quantos anos?	• quantos anos você tem?
$	Corresponde ao final da string, ou final de uma linha, caso o texto tenha múltiplas linhas.	!$	• Olá! • Tudo bem?!	• Olá.

()	Parênteses são usados tanto para agrupamento quanto para captura.	ba(na){2}	• banana • bananada • Doce de banana é bom.	• abanar • cabana

Observação: para escrever uma expressão regular que reconheça caracteres que normalmente teriam um significado especial (e.g., ., ?, (,), $), basta escapar estes caracteres com uma barra invertida, \. Por exemplo: a expressão que horas são\? reconheceria a string "Pode me dizer que horas são?", porque o ponto de interrogação está devidamente escapado.

Uso prático de expressões regulares

Muitas linguagens de programação e ferramentas possuem suporte à interpretação de expressões regulares. A maioria das linguagens de programação populares possui mecanismos nativos para a manipulação e uso de expressões regulares. Além disso, muitas ferramentas de linha de comando, como find, grep, sed e awk são comumente usadas com regexes.

O conceito de expressão regular é muito poderoso e é uma ferramenta essencial para resolver diversos problemas práticos. Por outro lado, para alguns problemas, a expressão regular correta pode ser bastante complexa e, sem a devida documentação, impossível de ser compreendida mais tarde—veja este exemplo da expressão regular oficial para reconhecer endereços de email em diversas linguagens.

Para encerrar, uma citação de Jamie Zawinski, co-fundador do Netscape e do Mozilla:

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Exercícios

Escreva expressões regulares para reconhecer os padrões descritos abaixo. Leia o enunciado até o final antes de começar.

Número do exercício	Descrição	Exemplos válidos	Exemplos inválidos
2.1	Um número USP (7 ou 8 dígitos), sem traços, pontos ou qualquer sinal de pontuação entre os dígitos.	• 7581993 • Meu número USP é: 10336827.	• 7581-993

2.2	Número de telefone (fixo ou celular), com DDD entre parênteses, código do país precedido por "+" e espaços. Sem traços.	● +55 (11) 999994321 ● +55 (51) 36975544	● 55 (11) 99999 ● +55 (11) 9-9999-4321
2.3	Linhas de um arquivo que começam com um número de até dois dígitos, seguido por parênteses e por uma frase qualquer. A linha deve terminar com um ponto de interrogação.	● 1) Qual o seu nome? ● 01) Qual o seu nome? E a sua idade?	● 01) Escreva o seu nome. ● (01) Escreva o seu nome.
2.4	Um nome próprio, com as seguintes restrições: ● No mínimo 2 nomes (nome e sobrenome) e no máximo 3. ● Todos os nomes devem começar com letra maiúscula (e este é o único lugar em que letras maiúsculas podem aparecer) ● O nome do meio, se presente, pode ser abreviado. A abreviação é a primeira letra do nome (maiúscula) seguida por um ponto ● Todos os nomes devem ter pelo menos 3 letras (a não ser que seja o nome do meio abreviado)	● Alan Mathison Turing ● Ada Lovelace ● Richard M. Stallman	● turing ● richie stallman ● LinuS Torvalds ● Edsger-Dijkstra ● Ed Dijkstra

Observações

- Nenhum destes exercícios tem uma resposta única. Dado um determinado padrão, é comum que existam várias expressões regulares que o reconheçam. Por exemplo, uma sequência de dois dígitos pode ser definida pelas expressões \d\d, \d{2}, [0-9]{2}, [0123456789][0123456789], etc.

- Você pode usar o site regex101 para testar e entender suas expressões regulares. Utilize os exemplos válidos e inválidos fornecidos no enunciado, e escreva a sua expressão lá, até ficar satisfeito com o resultado.

Entrega

Entregar um arquivo zip com nome lab2-atv4-XXXXXX.zip contendo um único arquivo chamado regex.txt.

O arquivo deve conter uma linha para cada exercício da tabela acima, começando com o número do exercício, seguido de parênteses ()), 1 espaço e a expressão regular que você criou para resolver o problema.

Exemplo:

2.1) \b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b
2.2) (19|20)\d\d
2.3) ^\s*#.*$
2.4) /*.*?*/