VOL. 5
2023

Revista HACKOONSPACE

Projeto Hackerspace
UFSCar Sorocaba

Apresentação dos artigos e projetos realizados na edição 2023 do HackoonSpace

Universidade Federal de São Carlos
Campus Sorocaba

PROEX
Pró-Reitoria de Extensão

Revista Hackoonspace

Vol. 5

Projeto Hackoonspace
2023

Apresentação

O Projeto Hackoonspace é um projeto de extensão promovido pela UFSCar Campus Sorocaba e realizado na forma de encontros que discutem a contracultura hacker, apresentando personagens, acontecimentos, aspectos socioculturais, artefatos e atividades com a finalidade de que os participantes sejam expostos à contracultura em questão e desenvolvam um projeto ou material acerca da mesma.

A principal intenção do projeto é desmistificar o conceito e figura do hacker, enquanto proporcionando um espaço de exposição, produção e compartilhamento de conteúdo que se configura como dentro da contracultura de forma que participantes de diferentes níveis de conhecimento técnico possam participar.

Esta revista tem como objetivo divulgar os trabalhos desenvolvidos pelos alunos participantes do Projeto Hackerspace no ano de 2023. Esperamos que esta revista possibilite a difusão dos conhecimentos adquiridos na atividade para a comunidade em geral. Gostaríamos de agradecer todos os alunos que contribuíram com artigos e projetos para esta edição da revista.

Organização e edição:
Leonardo Valerio Morales
Fernando Favareto Abromovick
Joatan da Silva Marques
Vinícius Carvalho Venturini

Supervisão:
Gustavo M. D. Vieira

Conteúdo

Conteúdo iii

I Artigos 1

1 Monitoramento de Redes de Computadores para aplicações *Cloud Gaming* 2
Carlos Henrique de França Marques

2 Pathfinding: Uma introdução teórica à consciência artificial 12
Breno da Costa Caricchio Aguiar

3 Guia de *Rust* 19
Vitor Kenzo Fukuhara Pellegatti

4 *OpenBSD*: Um sistema operacional livre focado em Segurança e Simplicidade 63
José Gabriel de Oliveira Santana

5 Compressão: O Que Torna a Tecnologia Moderna Viável 67
Ryan Guerra Sakurai

6 Decodificando as Linguagens de Programação 75
Ryan Guerra Sakurai

II Projetos 86

7 upGrade 87
Mauricio Cândido de Souza

8 P3 APP: Aplicação web para gerenciamento de tempo e tarefas 89
Pedro Gonçalves Correia
Pedro Enrico Nogueira Barchi
Pedro Henrique Alves de Araujo Silva

9 Dashboard Carteira de Investimento 92
Alex Sandro Momi Junior

10 BCC-HUB : Plataforma de Hospedagem Web Autogerenciada da Computação **95**
Rafael Gimenez Barbeta

Parte I

Artigos

Capítulo 1

Monitoramento de Redes de Computadores para aplicações *Cloud Gaming*

Carlos Henrique de França Marques

Acesse o artigo na íntegra clicando aqui

1.1 Introdução

Cloud Gaming é um dos principais serviços conectados à Internet atual. Poder jogar através da Nuvem possibilitou novas oportunidades e facilidades aos jogadores, uma vez que o jogo não roda em seu computador, não é necessário que ele tenha ótimas configurações para poder jogar.

Assim, essa modalidade ou forma de jogar está se tornando cada vez mais popular no Brasil e em todo o mundo. Porém, para que a qualidade de jogatina experienciada pelos jogadores seja boa e satisfatória, é necessário que a conectividade entre o dispositivo usado para jogar e os servidores provedores desses serviços seja muito eficaz, onde não pode haver muitas falhas e atrasos entre essa comunicação. Por exemplo, a conexão precisa ser de alta velocidade para que seus movimentos sejam processados e executados no jogo em tempo real, ou caso contrário haverá *input lag* (atraso de resposta à entrada). Outro problema que pode ocorrer é uma perda de qualidade de imagem durante a *gameplay*, devido a oscilações na rede.

Tendo em vista essa problemática, se torna necessário elaborar possíveis soluções para mitigar situações desagradáveis e melhorar a qualidade da experiência dos jogadores ao utilizar *Cloud Gaming*. Pensando nisso, um conceito que pode habilitar essas soluções é o de *Software Defined Networking* (SDN) [1], ou Redes Definidas por Software. Esse paradigma de redes de computadores separa o plano de controle do plano de encaminhamento de dados, centralizando as funções de rede e podendo programá-las por software, e assim podem ser criadas regras de encaminhamento e processamento de dados dentro de dispositivos de rede especializados, que são programáveis.

Assim, por meio dessa abordagem é possível criar formas de analisar a rede ou a conexão enquanto está sendo jogado por *Cloud Gaming*. E essa análise pode ser útil para criar soluções eficazes para atingir e estabilizar uma boa experiência de *gameplay*.

Com isso, estou realizando um projeto em conjunto com o professor Fábio Verdi e a equipe do seu Laboratório de Pesquisa (LERIS) [2], em que o objetivo é monitorar as filas de equipamentos de rede, durante o uso do *Cloud Gaming*. Para isto, iremos utilizar a linguagem de programação P4 (*Programming Protocol-Independent Packet Processors*) [3], que é de uso específico para a programabilidade de dispositivos de rede. E mais especificamente, será aplicada a abordagem INT (*InBand Network Telemetry*) [4], que nos permitirá medir a ocupação dos *buffers* desses dispositivos.

1.2 Contextualização

1.2.1 - *Cloud Gaming*

Os videogames são uma das principais formas de entretenimento do mundo contemporâneo. Com infinitas possibilidades de jogos, quase todo mundo já teve a experiência de se divertir jogando, e além disso, diversas pessoas encontraram nesses jogos sua profissão, seja como criadores de conteúdo ou jogadores profissionais.

Contudo, com a evolução dos jogos ao passar dos anos, eles passaram a demandar cada vez mais processamento gráfico e computacional, além de mais memória e espaço em disco. Isso se torna um limitante, uma vez que os computadores que acompanham essas necessidades dos games e possuem componentes tão poderosos não são baratos, pelo contrário. Assim, esse problema vira um obstáculo para o acesso aos jogos, uma vez que muitas pessoas não têm poder financeiro suficiente para adquirir consoles ou computadores modernos. Com isso em mente, vamos para a ideia do *Cloud Gaming*.

O *Cloud Gaming* é uma outra forma de jogar. Diferentemente do modo tradicional, onde os jogos rodam na máquina do usuário, e a conexão com servidores é somente para conectar os jogadores em *games multiplayer* online; nesta nova abordagem os jogos são totalmente executados em provedores remotos, fazendo com que todo o processamento fique concentrado na nuvem. Dessa forma, o computador do usuário assume apenas a responsabilidade de se conectar a esses servidores, transmitindo as entradas do jogador e exibindo a imagem do jogo na tela. Essa modalidade de jogo possibilitou que jogos mais exigentes sejam jogados em computadores menos potentes, uma vez que a execução do jogo não ocorre localmente neles.

Diante dessa, entre outras, motivação, o *Cloud Gaming* está cada vez mais sendo apoiado e ampliado. Novos jogadores estão surgindo, assim como novos provedores. Algumas das principais plataformas são: *Xbox Game Cloud*, *Playstation Now*, *GeForce Now* e *Amazon Luna*.

No entanto, existe uma grande problemática nesse contexto. Enquanto o *Cloud Gaming* livra o dispositivo do jogador de ter alto poder de processamento, ele necessita de que a conexão entre a máquina usuário e o servidor que está provendo o serviço seja excelente. Isso é necessário para que a qualidade da experiência do jogador seja suficientemente boa, fazendo com que ele sinta como se o jogo estivesse rodando em seu computador.

Entre os requisitos que a conexão cliente-servidor precisa ter, os principais são: baixa latência, uma vez que os dados de entrada e saída precisam ser encaminhados e processados em tempo real, assim tendo uma grande fluidez e baixo tempo de resposta dos comandos do jogador; e conexão estável, onde não pode haver grandes perdas de pacotes de rede, assim não ocorrendo travamentos ou interrupções durante o jogo.

Deste modo, se faz necessário que existam formas ou soluções para melhorar e garantir uma boa conexão durante o uso de *Cloud Gaming*.

1.2.2 - Programabilidade em Redes de Computadores

O mundo conectado que conhecemos hoje existe graças à criação e evolução da Internet. É ela que possibilita a conexão entre pessoas e dispositivos por todo o globo, assim como o uso de serviços online como redes sociais, *streaming* de filmes e séries, e mesmo os jogos online e *Cloud Gaming*. E a Internet nada mais é que a rede das redes, onde inúmeros dispositivos (roteadores, *switches*...) interconectam as redes e os computadores entre si.

Dentro da área de redes de computadores, foi percebido que as estruturas estavam se tornando maiores, mais complexas e mais difíceis de administrar e controlar. Assim, surgiu a construção de uma nova abordagem de configuração de redes, chamada SDN (*Software Defined Network*) [1]. Esse paradigma consiste na separação de dois componentes essenciais das redes de computadores: o plano de controle, responsável pelo gerenciamento geral da rede, aplicação de

políticas e definição de rotas de encaminhamento de pacotes; e o plano de dados, que faz de fato o processamento e o encaminhamento dos dados. Com essa separação, o plano de controle, que antes era executado em dispositivos de rede espalhados, passou a ser atribuído à controladores centralizados, enquanto o plano de dados se mantém em roteadores e *switches*.

Esses aparelhos, uma vez que se tornaram independentes do plano de controle, ficaram mais flexíveis e gerenciáveis, e dessa forma, surgem os dispositivos de rede programáveis. Este novo tipo de hardware permite com que sejam definidas tarefas mais complexas e inovadoras, de como os pacotes são processados dentro do plano de dados, por meio da programação desses equipamentos. Entre as tarefas que podem ser executadas estão o controle de tráfego, criação de regras de processamento e encaminhamento, monitoramento de rede, entre outras.

A principal linguagem de programação para dispositivos de rede atualmente é o P4 (*Programming Protocol-Independent Packet Processors*) [3]. Essa linguagem, criada especificamente para o contexto da programabilidade em redes, permitiu com que fossem criadas implementações mais flexíveis e funcionais, uma vez que ela traz a não dependência de protocolos de rede específicos, além de possibilitar a execução em diversos modelos de equipamentos programáveis.

Com o surgimento e a evolução dos dispositivos mencionados, juntamente com o desenvolvimento da linguagem P4, surge também a capacidade de realizar a coleta de uma ampla gama de informações diretamente da rede. Isso inclui um monitoramento mais próximo de hardware de rede, atendendo a uma variedade de objetivos e aplicações.

1.3 O Projeto

1.3.1 - Ideia Geral

Tendo em vista esse contexto, foi decidido realizar um projeto de pesquisa que contribuísse para o avanço desse âmbito de soluções para melhora da qualidade de experiência dos jogadores de aplicações *Cloud Gaming*, por meio do paradigma e tecnologias de SDN e programabilidade em redes de computadores. Mais especificamente, o trabalho em desenvolvimento é o monitoramento de filas de dispositivos de rede, como os *switches* programáveis, a fim de coletar informações sobre a ocupação dessas filas durante o uso do *Cloud Gaming*.

Esse projeto está sendo desenvolvido em conjunto com o professor da UFSCar Fábio Verdi, e seu grupo de pesquisa na área de Redes de Computadores, o LERIS [2]. Eles estão e vão me auxiliar durante o processo, indicando os passos a serem tomados e ajudando com eventuais dúvidas e dificuldades.

1.3.2 - Preparação Inicial

Para realizar os experimentos, iremos utilizar, primeiramente, o *Raspberry Pi* [5], que é uma espécie de microcomputador, em formato de apenas uma placa lógica, mas que possui processador, memória RAM, placa de vídeo, assim como entradas USB, HDMI, interfaces física de rede (*Ethernet*) e não física (*WiFi*). Esse dispositivo foi criado principalmente para a educação na área de Computação, mas que também pode ser usado para outros propósitos, como pesquisa, sendo que ele foi escolhido para ser utilizado pela sua maior facilidade de uso, em relação aos próprios *switches* programáveis. A imagem 1 é uma foto de um Raspberry Pi.

Figura 1.1: Foto de um *Raspberry Pi* [6]

Para habilitar o *Raspberry Pi* para nosso pro-

pósito, foi feita nele a instalação do P4Pi [7]. O P4Pi é uma plataforma de código aberto, sendo baseada na placa do *Raspberry Pi* e é utilizada para o propósito de educação e pesquisa. Ele foi desenvolvido pelo mesmo grupo responsável pelo P4, e assim foi implementado para a utilização da linguagem. O processo de instalação do P4Pi consistiu em realizar o download da imagem da plataforma, disponível em sua *Wiki* [8] no *GitHub*; e então escrevê-la em um *microSD* por meio de um programa próprio [9] do *Raspberry*, que por fim é conectado no aparelho. Todo o tutorial da instalação [10] se encontra na *Wiki*.

Uma vez com o P4Pi instalado, o *Raspberry* cria uma rede *Wi-Fi*, e a partir dela podemos acessá-lo remotamente, utilizando o comando SSH [11]. Além disso, é importante ressaltar que o P4Pi executa uma emulação de um *switch* programável, o que permite utilizá-lo para nosso propósito.

Tendo o *Raspberry Pi* em mãos e configurado, partimos para alguns testes iniciais. Primeiramente temos que conseguir acessar a Internet por meio dele, ou seja, realizar uma conexão que tenha ele como caminho, para assim futuramente monitorar como os pacotes que passam por ele se comportam nas filas. Para isso, rodamos um programa de exemplo, que já veio instalado junto ao P4Pi, chamado *L2 Switch* [12]. Esse programa faz com que o *switch* emulado realize uma ponte entre suas portas virtuais, realizando o roteamento por endereço MAC. Além disso, é necessário rodar um *script* de configuração do P4Pi para conectar sua interface física *Ethernet* ao *switch* emulado. Assim, executando o *script* e ligando o *Raspberry* por cabo de rede à um roteador com acesso a Internet, podemos nos conectar a Internet ao entrarmos na rede *WiFi* do *Raspberry* com um *laptop* ou PC.

Com essa conexão funcionando, chegou a hora de testar um pouco o *Cloud Gaming*. O objetivo era saber se seria possível jogar em nuvem estando conectado ao *Raspberry*, já que, uma vez que a conexão entre ele e o notebook é por *Wi-Fi*, poderiam ocorrer muitos problemas ou travamentos durante o jogo, como os citados anteriormente. Para realizar esses testes, utilizei a plataforma *GeForce Now* [13], mas depois usei também o *Xbox Cloud Gaming* [14]. Instalei o aplicativo do *GF Now* no notebook e conectei minhas contas *Epic Games* e *Ubisoft Connect*. Assim tive acesso a alguns jogos, comecei a jogar. Testei os jogos *The Crew* e *Trackmania* em minha casa, e depois no laboratório do LERIS joguei *Fortnite*. Percebi que a *gameplay* estava boa, com poucos travamentos ou atrasos. Após isso, o próximo passo foi incrementar o cenário, utilizando dois *laptops* ao mesmo tempo para jogar. Dessa forma, fiz o teste, com um notebook utilizado por mim e outro por um amigo. A experiência continuou sendo aceitável, acontecendo somente um pouco mais de travamentos no começo da jogatina.

1.3.3 - Métricas de monitoramento e como coletar

Nesse tópico, vou explicar sobre a forma escolhida para a realização da coleta das métricas de monitoramento de filas, e como ela funciona.

A abordagem que está sendo utilizada é a *Inband Network Telemetry* (INT) [4]. Ela consiste na coleta de dados meta fornecidos pelos *switches* programáveis P4, ou seja, são dados sobre os próprios *switches*. Esses dados, chamados de *Standard Metadata* [15], podem ser acessados pelos programas P4 (da mesma forma que se acessa uma variável em linguagens gerais, como em C) e utilizados para diversos propósitos, como definir a porta de saída de pacotes, descobrir o tamanho de um pacote, o tipo de instância dele, etc.

Entre esses dados, o INT busca coletar aqueles relacionados às portas de ingresso e egresso (ou entrada e saída), ao tempo que um pacote levou para entrar e sair (tempo de processamento), e à profundidade da fila nos momentos que o pacote entrou e saiu da mesma. Todos esses são *Standard Metadata* fornecidos pelos *switches*:

- ingress_port - valor da porta onde o pacote ingressou no switch;

- egress_port - valor da porta onde o pacote vai sair do switch;

- egress_spec - valor onde costuma ser indicado a porta de saída para o processamento do switch;
- ingress_global_timestamp - o momento, em microssegundos, de quando o pacote chegou no processamento de ingresso, é calculado por um relógio próprio do switch, que inicia em 0 quando ele é ligado;
- egress_global_timestamp - o momento, em microssegundos, de quando o pacote começa o processamento de egresso, usa o mesmo relógio do ingress_global_timestamp;
- enq_timestamp - o momento, em microssegundos, que o pacote foi enfileirado pela primeira vez, também usa o mesmo relógio;
- enq_qdepth - profundidade de ocupação da fila, no momento que o pacote é enfileirado;
- deq_timedelta - tempo, em microssegundos, que o pacote passou na fila;
- deq_qdepth - profundidade de ocupação da fila, no momento que o pacote sai da fila.

Agora, se faz necessária uma forma de coletar esses dados e então poderem ser lidos e armazenados. Então, utilizando a abordagem INT, o que se faz é enviar um pacote especial ao *switch*, onde nele é escrito os dados, e então o pacote é enviado e recebido outro computador que lê as informações. Isso funciona por meio da criação de um novo cabeçalho de rede (como IP, *Ethernet*, TCP) nos pacotes a serem enviados. Assim são adicionados campos novos ao pacote, e neles são armazenados os dados citados anteriormente.

Figura 1.2: *Setup* clássico para utilização do *Inband Network Telemetry* (INT) [16]

A figura 1.2 ilustra o *setup* de laboratório clássico para a utilização do INT. Nele o *Host-1* envia um pacote (ou vários) para o *Switch-1*, onde são colhidos os primeiros dados, e então o pacote vai para o *Switch-2*, colhe as informações nesse switch, e por fim o pacote é recebido pelo *Host-2*. Porém, para o nosso caso, queremos colher os dados de apenas um *switch* (o P4Pi), e o mesmo precisa estar conectado a Internet para que possamos usar o *Cloud Gaming*. Então foi necessário criar um *setup* diferente.

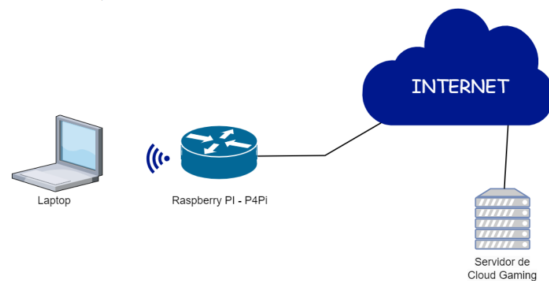

Figura 1.3: *Setup* modificado para se adequar às condições do projeto

A figura 2 mostra o *setup* utilizado para nossa coleta de dados. Nele temos um laptop, conectado ao P4Pi por *Wi-Fi*, que é onde é jogado o *Cloud Gaming*, e também é ele que envia os pacotes INT. Os pacotes são enviados até o P4Pi, onde as métricas desejadas são colhidas, e então o pacote volta pela mesma porta, sendo enviado novamente para a rede sem fio, e chegando até o laptop. E essa é a ideia de como coletar as métricas de monitoramento de filas para *Cloud Gaming*.

1.3.4 - Configurando *setup* para a coleta de dados

Tendo em vista todos esses conceitos e ideia do *setup*, chegou a hora de fazer as configurações necessárias para termos tudo isso funcionando. Primeiramente temos que ter em vista que, para a abordagem INT funcionar e conseguirmos coletar as métricas, precisamos ter um programa INT P4 (rodando no *Switch*/P4Pi) que acesse os *Standard Metadata* desejados e os armazene em cada pacote que chegar e possuir nosso cabeçalho INT especial. Além disso, é necessário também termos dois programas em Python [17]

(rodando no laptop), o *send* e o *receive*, em que o primeiro é responsável por criar e enviar nossos pacotes INT, e o segundo faz o recebimento e o processamento dos pacotes, como imprimi-los na tela ou armazená-los na memória. A figura 1.4 a seguir ilustra isso.

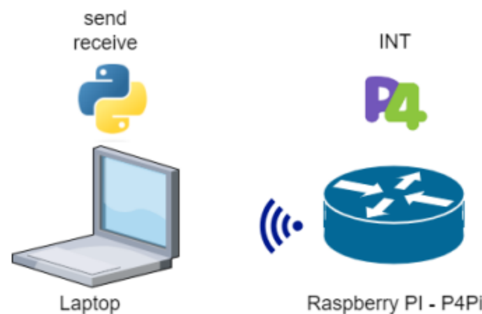

Figura 1.4: conexão dos programas *send* e *receive* com o programa INT P4 do *switch* programável

Tanto para o programa em P4 quanto para os em *Python*, eu consegui ter uma boa base inicial com um laboratório virtual de INT [16] já criado anteriormente por um membro do LERIS, o Leandro Almeida [18]. Nesse laboratório, ele fez todos os programas necessários, mas como eles foram feitos para o *setup* virtual do Leandro (o mesmo da figura 1), era necessário realizar modificações, para que eles funcionassem para o nosso caso.

Olhando primeiro para o código P4, nele foi necessário alterar a regra de roteamento dos pacotes. Antes estava implementado o roteamento por endereço IP, onde o *switch* decidia qual porta enviar cada pacote por meio de seu IP de destino. Eu alterei isso, uma vez que não queríamos usar esse tipo de roteamento, pois para isso funcionar com nosso *setup* conectado a Internet seria necessário implementar um protocolo complexo para saber qual porta enviar os pacotes. Então criei uma regra que simplesmente envia o pacote para a porta diferente da onde o pacote veio, ou seja, se o pacote chegou por *Wi-Fi*, encaminha para a interface *Ethernet*, e vice-versa. Essa regra é aplicada para todos os pacotes, exceto aqueles que possuem nosso cabeçalho especial INT. Para esses criei uma outra regra, também bem simples, que faz o contrário dos pacotes comuns, e envia o pacote de volta para a mesma porta que ele veio. Dessa forma, podemos acessar a Internet normalmente, inclusive usar o *Cloud Gaming*, enquanto nossos pacotes de monitoramento podem colher as informações e voltar para o laptop, que as processa.

Sobre os programas em *Python*, tive que fazer somente modificações bem pequenas, como indicar neles para enviar e receber pacotes pela interface *Wi-Fi* do laptop, entre outros pequenos detalhes. Assim os *scripts* [19] estavam prontos para uso. Eles estão no *GitHub*.

Tendo os códigos prontos, vou explicar agora como fiz para rodá-los/instalá-los no *setup*. Falando primeiro sobre os *Python*. Como o laptop que estou utilizando está com o *Windows* 10 instalado, optei por usar o *PyCharm* [20], que é uma IDE tranquila de usar e própria para *Python*, onde é possível editar e executar os códigos facilmente. Para isso, primeiramente foi necessário instalar o próprio *Python* na máquina, pelo site oficial, e depois instalei o *PyCharm*. Além disso, foi preciso também importar as bibliotecas necessárias, que estão sendo utilizadas nos programas, como o *Scapy* [21], o que foi feito dentro da IDE. Feito isso, já estava sendo possível rodar o programas *send* e *receive*.

Passando agora para o código P4, era preciso instalá-lo no P4Pi, para que o *switch* virtual dele pudesse executar aquelas regras explicadas anteriormente. Assim, eu criei um projeto próprio dentro da pasta do *switch*, onde criei um arquivo *int.p4*, e por meio de alguns comandos no terminal indiquei para o P4Pi compilar e executar o meu código no *switch*.

Com todos esses passos concluídos, o *setup* estava pronto para colher as métricas. Uma vez que o notebook estava conectado na rede do P4Pi, e o aparelho plugado na Internet, bastava executar os códigos *Python*, e assim podia ver as métricas aparecendo na saída do *receive*, conforme os pacotes INT enviados iam chegando. A imagem 1.5 é uma *screenshot* que mostra como os pacotes

aparecem na tela.

```
    \options   \
###[ nodeCount ]###
        count     = 1
        \INT       \
         |###[ InBandNetworkTelemetry ]###
         |  switchID_t= 1
         |  ingress_port= 0
         |  egress_port= 0
         |  egress_spec= 0
         |  ingress_global_timestamp= 524561956512
         |  egress_global_timestamp= 524562011075
         |  enq_timestamp= 575946796
         |  enq_qdepth= 5
         |  deq_timedelta= 54171
         |  deq_qdepth= 5
```

Figura 1.5: Esquema geral dos pacotes recebidos e suas métricas

1.3.4 - Armazenando as métricas em um Banco de Dados

Com o *setup* funcionando, podíamos ver as métricas de monitoramento na tela do laptop, o que era interessante, mas faltava uma forma de guardá-las em memória, para que pudessem ser analisadas posteriormente.

Dessa forma, o próximo passo do projeto foi configurar o envio das informações INT que chegam dos pacotes para um Banco de Dados Time Series, que é um tipo de banco de dados especializado e otimizado para armazenar informações relacionadas ao tempo, como acessos de login em um site, resultados de monitoramento geográficos e ambientais, entre outros. Esse tipo de sistema é o ideal para o nosso caso, pois o objetivo era enviar um pacote INT por segundo durante a jogatina com *Cloud Gaming*, para termos um grande volume de dados e assim poder analisar o comportamento das filas durante um período de tempo.

A plataforma de banco de dados temporais escolhida foi a *InfluxDB* [22]. Ela foi indicada para mim por já ter sido usada anteriormente pelos membros do LERIS. Seu funcionamento é totalmente online, armazenando os dados em servidores, que podem ser acessados diretamente pelo navegador em seu site. Para utilizar a plataforma, primeiramente tive que criar um perfil, e então criar o banco de dados (no *InfluxDB* é chamado de *bucket*) onde seriam armazenadas as métricas coletadas.

Assim, restava apenas configurar o envio dos dados até o banco. Para isso, tive que modificar o código *receive*, para que ele, além de imprimir as informações na tela, se conectasse ao *InfluxDB* e inserisse as métricas coletadas de cada pacote. Toda essa parte de conexão e escrita no banco de dados é feita com a utilização de uma biblioteca própria do *InfluxDB*.

Feitos esses passos, nosso *setup* estava totalmente pronto para coletar as métricas de monitoramento de filas enquanto jogo em *Cloud Gaming*. A figura 1.6 ilustra o ambiente final.

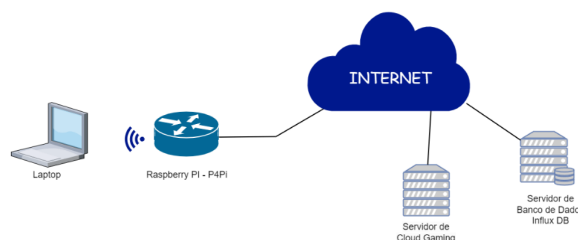

Figura 1.6: *Setup* final, após considerados todos os detalhes do projeto

1.3.5 - Primeiros resultados monitorando *Cloud Gaming*

Após todo esse processo, chegou a hora de realizar o primeiro experimento. Então, resolvi fazer o monitoramento das filas de rede por uma hora enquanto jogo *Fortnite* [23]. Para isso, utilizei a plataforma *Xbox Cloud Gaming*, que permite jogar esse jogo gratuitamente por mais de uma hora sem interrupções.

Assim, fiz as preparações iniciais, ligando o P4Pi na Internet, abrindo sua porta cabeada para o *switch* virtual, conectando o notebook em sua rede *Wi-Fi*, e executando os programas *Python send* e *receive* (o programa INT P4 já executa automaticamente ao ligar o P4Pi). O *send* foi

configurado para enviar um pacote por segundo. Por fim, bastou abrir o *Fortnite* na *Xbox Cloud* e jogar. Passado uma hora, parei de jogar e interrompi o *receive*. O experimento foi realizado no laboratório LERIS.

Após isso, entrei na plataforma do *InfluxDB* e baixei os dados colhidos e os coloquei em uma Planilha *Google* [24], onde pude criar 2 gráficos com os resultados.

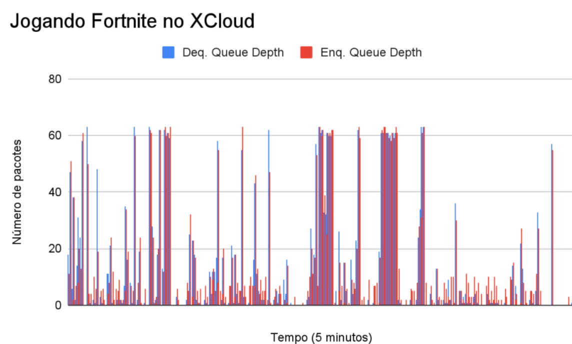

Figura 1.7: Gráfico 1: Número de pacotes *Enq. Dequeue Depth* e *Deq. Dequeue Depth* por segundo, durante 5 minutos

O gráfico 1 (figura 1.7) mostra os valores atingidos de *Enq. Dequeue Depth* e *Deq. Dequeue Depth* a cada segundo durante um período de 5 minutos de jogatina, o que demonstra o nível de profundidade da fila em número de pacotes nesse intervalo. Decidi representar somente 5 minutos no gráfico para ter uma melhor visualização dos resultados. O valor máximo atingido foi 63 para ambos os valores, e as médias foram 13,5 o primeiro e 12,1 para o segundo.

A partir desse gráfico já é possível perceber que ocorreu uma grande oscilação da ocupação da fila, em que a maior parte dos valores se manteve entre 10 e 30 pacotes, e com alguns picos chegando perto ou mais de 60 pacotes.

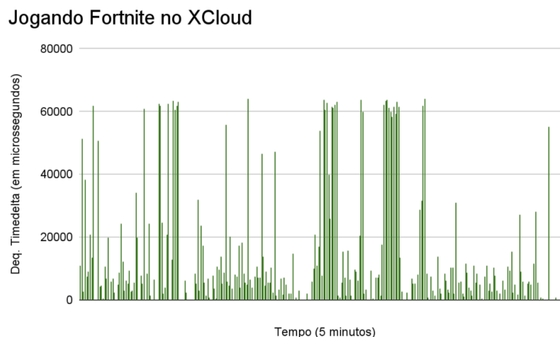

Figura 1.8: Gráfico 2: Número de pacotes *Deq. Timedelta* por segundo, durante 5 minutos

O gráfico 2 (figura 1.8) ilustra os valores de *Deq. Timedelta* a cada segundo, durante o mesmo período de 5 minutos de jogatina usados no gráfico anterior, o que demonstra o tempo que cada pacote INT passou na fila, em microssegundos, nesse intervalo. O valor máximo atingido foi 64123, e a média foi 13937,4.

Ao olhar para esse gráfico, podemos observar que, assim como a ocupação, o tempo que os pacotes ficaram na fila também teve bastante oscilação, com a maior parte dos valores se mantendo entre 10000 e 30000, e tendo alguns picos de mais de 60000.

Comparando agora os dois gráficos e os valores obtidos, fica evidente uma correlação forte entre o nível de ocupação e o tempo de permanência do pacote na fila. É perceptível que nos momentos de pico de ocupação, há um correspondente aumento no tempo de espera. Além disso, é interessante notar uma relação aproximada de 1:100, o que significa que a adição de cada pacote na fila resulta, em média, em um aumento de cerca de 100 microssegundos no tempo que o pacote passa na mesma.

1.4 Conclusão e Próximos Passos

Ao final deste semestre, posso dizer que consegui realizar um grande trabalho e avanço. Durante esse período, fui capaz de transformar a ideia inicial, de construir um ambiente de monitoramento de filas de dispositivos de rede, possibilitando a

coleta de informações relacionadas à ocupação dessas filas durante o uso de aplicações *Cloud Gaming*, e torná-la realidade.

Para isso, construí um *setup* com o dispositivo *Raspberry Pi*, juntamente com a plataforma P4Pi, que viabilizou ter um *switch* programável emulado fácil de usar e configurar dentro do ambiente. Apliquei a abordagem *Inband Network Telemetry*, que possibilitou a coleta das métricas desejadas, por meio de envio, processamento e recebimento de pacotes. Com a coleta funcionando, implementei também uma forma de armazenamento das informações, com o uso da plataforma de banco de dados temporais *InfluxDB*.

Com o ambiente pronto, fiz um primeiro experimento de monitoramento, coletando as métricas de ocupação das filas por uma hora, enquanto jogava *Fortnite* com *Xbox Cloud*. Assim pude ter as primeiras conclusões e resultados a partir do monitoramento realizado e dados colhidos, como a observação de picos de ocupação e a relação direta com o tempo de permanência dos pacotes nas filas.

No começo do semestre, tínhamos o planejamento de realizar a coleta e monitoramento de outros jogos, e utilizar um outro equipamento chamado *Intel Tofino* [25], um *switch* programável de fato, que nos dá mais possibilidades o *Raspberry Pi*/P4Pi. Mas posteriormente foi decidido que isso seria um trabalho para o próximo semestre.

Assim, como próximos passos desse projeto, além de utilizar outros jogos e o *switch Tofino*, o objetivo será realizar uma coleta de dados maior, seguindo uma metodologia científica para coleta e análise de dados. Ademais, planejamos criar uma grande base de dados, que será disponibilizada à comunidade, para que possa ser utilizada como auxílio na criação de soluções para melhora da qualidade de experiência dos jogadores de aplicações *Cloud Gaming*, que é o grande objetivo deste projeto.

O trabalho realizado neste semestre foi muito importante e gratificante para mim. Com ele, aprendi e me aprimorei em diversos conceitos e técnicas, como o meu conhecimento em aplicações *Cloud Gaming* e suas propriedades, em redes de computadores e programabilidade, em programação no geral, uso de linha de comandos, banco de dados, entre outros. Dessa forma, tive uma grande experiência, que com certeza foi importante para meu desenvolvimento e futuro acadêmico e profissional.

1.5 Bibliografia

[1] Software-defined networking (sdn) definition. URL: `https://opennetworking.org/sdn-definition/`.

[2] Leris. URL: `https://leris.dcomp.ufscar.br/`.

[3] P4 – language consortium. URL: `https://p4.org/`.

[4] In-band network telemetry (int) dataplane specification. URL: `https://p4.org/p4-spec/docs/INT_v2_1.pdf`.

[5] Raspberry pi foundation. raspberry pi — teach, learn, and make with raspberry pi. URL: `https://www.raspberrypi.org/`.

[6] Case raspberry pi 2 3 b+ preto. URL: `<ttps://www.autocorerobotica.com.br/case-raspberry-pi-2-3-b-preto`.

[7] P4pi - getting started. URL: `https://github.com/p4lang/p4pi`.

[8] P4pi - wiki. URL: `https://github.com/p4lang/p4pi/wiki/`.

[9] Raspberry pi. raspberry pi os. URL: `https://www.raspberrypi.com/software/`.

[10] Installing p4pi. URL: `https://github.com/p4lang/p4pi/wiki/Installing-P4Pi`.

[11] ssh command usage, options, and configuration in linux/unix. URL: `https://www.ssh.com/academy/ssh/command`.

[12] Simple l2 switch with bmv2 target. URL: `https://github.com/p4lang/p4pi/wiki/Example-%231---Simple-L2-switch-with-Bmv2-target`.

[13] Nvidia geforce now. URL: https://www.nvidia.com/pt-br/geforce-now/.

[14] Xbox cloud gaming (beta) em xbox.com. URL: https://www.xbox.com/pt-br/play.

[15] behavioral-model/simple_switch.md at main · p4lang/behavioral-model. URL: https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.

[16] L. ALMEIDA. Lab int-p4. URL: https://github.com/leandrocalmeida/InbandNetworkTelemetry-P4.

[17] Python. python. URL: https://www.python.org/.

[18] leandrocalmeida - overview. URL: https://github.com/leandrocalmeida.

[19] C. FRANÇA. carlos-hfm/p4pi-leris. URL: https://github.com/carlos-hfm/monitoramento-redes-CG.

[20] Pycharm: o ide python da jetbrains para desenvolvedores profissionais. URL: https://www.jetbrains.com/pt-br/pycharm/.

[21] Scapy. URL: https://scapy.net/.

[22] Influxdb | real-time insights at any scale. URL: https://www.influxdata.com/influxdb/.

[23] Fortnite. URL: https://www.fortnite.com/.

[24] Google sheets: Free online spreadsheets for personal use. URL: https://docs.google.com/spreadsheets/.

[25] Programabilidade p4 do intel® tofino tm 2 com mais largura de banda. URL: https://www.intel.com.br/content/www/br/pt/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

Capítulo 2

Pathfinding: Uma introdução teórica à consciência artificial

Breno da Costa Caricchio Aguiar

Acesse o artigo na íntegra clicando aqui

2.1 Introdução

O presente trabalho se empenha em compreender uma análise simplificada, entretanto, abrangente e principalmente teórica sobre os funcionamentos modulares da mente artificial e suas articulações. Desta forma, neste artigo de caráter introdutório, busca-se entender os limites e restrições modulares pertencentes ao nosso funcionamento intelectual (o funcionamento humano) e como isso pode ser aplicado em funcionamentos intelectuais artificiais. Primeiramente, serão apresentadas todas as modulações utilizadas por uma IA para uma formulação responsiva e posteriormente, as dependências neurais utilizadas para o alcance da IA dentro de níveis linguísticos, desde uma database léxica até propriedades semânticas e pragmáticas. Na segunda parte deste artigo, será analisado em seu final, a possibilidade de uma IA poder despertar uma consciência através de ensinamentos principalmente pragmáticos, onde esta poderá ser contemplada com arbítrios humanos como a tomada de decisões, gostos pessoais e outras diversas humanidades.

Neste começo, é necessário introduzir ideias consideráveis para a produção de tal consciência, no entanto em seu final, buscam-se muitas outras respostas. Na intuição humana de formular artificialidades similares ao funcionamento humano, é possível que estejamos prontos a nos deparar com certas estranhezas - a modulação artificial pode ser diferente da nossa e ainda dentro destas modulações, os aprendizados podem nem ser necessários, já que serão apresentadas *databases* às IAs.

Neste artigo, não deverá haver nenhum tipo de aspecto que se prenda a alguma corrente teórica-metodológica e/ou epistemológica, devido a necessidade de reconhecer peças individuais de cada corrente como uma peça-chave deste estudo, todavia, a interface cognitiva deve se comunicar muito profundamente com a corrente funcionalista devido não só ao *output* comunicativo que a consciência deve obter - segundo Denes e Pinson (1993) em "*The Speech Chain*", conseguir estabelecer uma "cadeia de fala" articulada e definida - mas também pela interação *input/output* que a linguagem possui para o alcance da consciência.

2.1.1 - Introdução aos módulos cognitivos humanos

Talvez seja improvável e extremamente simplista modular a cognição humana em números, mas definitivamente é assim que se define o cognitivo humano diante de algumas linhas epistemológicas. Levando em consideração principalmente o fator da adaptabilidade humana, a cognição humana se adequa ao ambiente, devido aos

diferentes estímulos e as demandas ambientais, se adequa a necessidade de aprendizado e de desenvolvimento, pois o indivíduo como ser social e adaptável e portanto, à deriva de influências sociais e influências ambientais, possui características interativas que o permitem se desenvolver através de parâmetros lógicos, como visto em diversos estudos da aquisição de linguagem: o ser humano nasce e possui uma absorção linguística profunda, a cognição na faixa etária de 3-12 anos se utiliza de parâmetros até de fato fixar sua língua nativa, portanto, através da influência do ambiente e dos seus pais (meio social), a criança capta os signos, as significantes e os significados de maneira natural, atribuindo tais conceitos naturalmente, desta forma, a aquisição de linguagem se apresenta como um exemplo desta interação do indivíduo com o meio.

O ser humano, em seu desenvolvimento empírico, aprende a diagnosticar e resolver problemas através do módulo do processamento das informações e apresenta diversas formas diferentes de se resolver um único problema, a solução de problemas também se apresenta como outro módulo humano. Por fim, o ser humano compreende uma capacidade de interação social que pode ser considerado fisiológico e também sociológico (seguindo as ações sociais de Max Weber), já que são as interações sociais que formulam a geração de descendentes. Vale ainda destacar a conexão entre os módulos, sendo todos eles dependentes uns dos outros, não é possível visualizar uma solução de problemas sem um processamento íntegro de informações e nem mesmo uma interação social sem o aprendizado e o desenvolvimento, a ética e a moral humana se destaca através da junção dos dois últimos exemplos dados.

2.1.2 - Estrutura de funcionamento e comparações artificiais

É necessário enfatizar que as modulações atribuídas às inteligências artificiais possui sua semelhança às modulações humanas explicadas introdutoriamente na parte anterior, o que é perfeitamente compreensível, já que tais inteligências foram produzidas e inferidas por um conhecimento prévio estabelecido por seres humanos. Segundo Turing (1950): "As máquinas não pensam como seres humanos pois uma máquina é diferente, logo ela pensa diferentemente. A pergunta intrigante seria: só porque algo pensa diferente, significa então que ela não pensa?", logo, é possível levar em consideração que há diferenças em suas mecânicas de funcionamento e na sua estrutura, para este estudo, iremos considerar ambas as diferenças para fazer as devidas comparações.

Uma das diferenças mais interessantes é, justamente, a consulta de informações para a organização de um conceito, ou seja, os seres humanos normalmente adquirem suas informações através de suas experiências cotidianas, o que deve incluir também seus estudos e a teoria adquirida, desta forma, se organizam informações que acabam por concretizar conceitos. Tais conceitos são utilizados para a veiculação de mais informações, a produção de outros conhecimentos, mas este é o principal motivo para o desenvolvimento intelectual. Para as máquinas, este mecanismo funciona, principalmente, em seu módulo de PLN (Processamento de Linguagem Natural) através de módulos menos abstratos: são codificados bancos de dados ricos em informações - esses bancos de dados são de corpus selecionados pelos desenvolvedores da IA em questão - e desta forma, a IA pode formular seus conceitos livremente, vale lembrar que há vários bancos de dados diferentes, como o banco de dado léxico, que fornece os signos necessários para a máquina atingir alguma forma de expressão compreensível pelos falantes de línguas naturais.

Tais conceitos são formulados através de analisadores sintáticos, semânticos e posteriormente, discursivos e pragmáticos, o que deve contribuir para espaços de enunciação mais humanos e a interpretação seja relativamente precisa, pois não é possível sintetizar informações em conceitos se não houver uma direção cognitiva referente à humana, sem um analisador pragmático, a máquina não seria capaz de tomar decisões e se ater a arbítrios. Ainda neste tópico, é discutível as sinapses das inteligências em termos de estrutura de funcionamento; A estrutura de funcionamento cognitivo das máquinas são extremamente similares ao modelo de PLN padrão e como dito

anteriormente, são estabelecidos e mantidos através de sinapses artificiais que conectam analisadores à bancos de dados. Redesenhando a padronização de PLN e adaptando-a para uma esquematização geral de funcionamento mecânico (exemplo), temos:

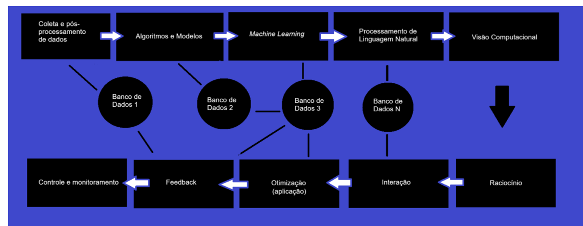

Figura 2.1: Esquema geral do funcionamento de PLN

Nessa breve esquematização, os quadrados representam os módulos, e os círculos, os bancos de dados. Novamente, deve ser enfatizado o fato de que este é uma esquematização que procura demonstrar como os processos cognitivos se consomem em sua própria estrutura de pensamento. Para cada necessidade, cria-se um banco de dados diferente e esta criação não deve se limitar - por isso um "Banco de Dados N", sendo N uma variante que diz respeito ao número que representa a quantidade de bancos de dados - portanto, o primeiro módulo coleta dados/informações e infere um processo de processamento para que os algoritmos possam utilizar tais informações dependendo de sua interação com um usuário. Os algoritmos são escritos no código de execução das máquinas para que possam cumprir com seus papéis cognitivos, estes são responsáveis por grande parte do material essencial e intelectual da máquina, desde todo seu aprendizado através do *Machine Learning* até o possível desenvolvimento de um arbítrio perfeitamente funcional visível em alguns outros módulos como a interação e o raciocínio. No Processamento de Linguagem Natural, são produzidas as capacidades linguísticas, competências de linguagem e formulações que também devem se conectar a outros módulos como a visão computacional, afinal uma máquina não pode produzir significados e ter uma visão sem um desenvolvimento adequado de sua linguagem, isso também deve envolver alguns modos conseguintes, como o raciocínio geral e a interação com o usuário, logo, uma interação social. O módulo de otimização será explorado no tópico de aplicações, no entanto, este é responsável pela otimização das informações dadas e o *feedback* deve retornar possíveis soluções ao desenvolvedor da IA em questão. Em seu último módulo, o "Controle e Monitoramento" existe como uma segurança a mais implementado ao código da IA, buscando solucionar os erros e desenvolver uma capacidade cognitiva mais precisa.

2.2 Análise

2.2.1 - O módulo de processamento de linguagem natural (PLN)

Como dito anteriormente, este módulo é responsável pela implementação da linguagem e dos recortes linguísticos fundamentais para as IA. Este módulo possui uma relativa importância devido ao questionamento inicial de tomar um passo à frente das tecnologias que envolvem as inteligências artificiais de fato, pois este processamento é o responsável pela produção de uma possível linha de pensamento através do desenvolvimento evolutivo da linguagem. A estrutura de funcionamento têm seu processo similar ao processo de PLN:

Figura 2.2: Modelo padrão de PLN

Neste modelo padrão do processamento, a sentença é tida como o *input*, cada pequeno retângulo representa um analisador que pertence a um nível linguístico distinto e relacionável e cada pequeno círculo representa um banco de dados utilizado. A "Gramática" como um sistema é internalizada para um banco de dados o qual a IA pode adquirir suas informações gramaticais, esta é disponibilizada em seu nível sintático. Se apresenta uma sentença, o analisador léxico se organiza através das informações do banco de dados "Léxico", em seguida, são atribuídos traços aos *tokens* - cada pequeno constituinte do *input*, ou seja, da sentença inicialmente proposta - essa atribuição deve organizar e portanto otimizar a sentença para os próximos analisadores. Em sua próxima etapa, a IA usa de uma gramática (apresentada através de um banco de dados) e também usa do léxico para produzir estruturas sintáticas coesas, após isso, o analisador semântico verifica atributos como sentidos e significados dentro da estrutura sintática previamente construída e retorna uma representação semântica dotada de sentido, se utilizando ainda de um banco de dados específico chamado de "Modelo de Domínio" que contém informações específicas sobre um objeto, o analisador discursivo deve estender a representação semântica mantendo todas as outras correções feitas pelos analisadores e por fim, o analisador pragmático usa o "Modelo de Usuário" para verificar aspectos reais do usuário da IA em questão, possibilitando diferenciações arbitrárias e compreensão profunda de seu(s) usuário(s), retornando uma representação completa do conteúdo e dos objetivos comunicativos.

2.2.2 - A dicotomia significante e significado de F. Saussure

Em diversos contextos e momentos, o ser humano propôs diversos estudos que posteriormente foram intitulados como fazendo parte das ciências, o que é chamado hoje de científico. Nos recortes históricos, tivemos a inauguração da Linguística como uma ciência de fato, através das diversas dicotomias apresentadas por Ferdinand Saussure com o "Curso de Linguística Geral", obra utilizada por vários linguistas de diversas correntes epistemológicas dessa ciência nos dias de hoje. Neste artigo, a dicotomia "Significante e Significado" será discutida com o intuito de explicar o fundamento por trás do signo linguístico e desta forma, entender a relação das análises linguísticas propostas tanto pelo ser humano quanto pelas IAs.

Primeiramente, Saussure procura nomear os constituintes desse estudo: "O signo linguístico une não uma coisa à uma palavra, mas um conceito à uma imagem acústica" (SAUSSURE, 1997, p.80). O signo linguístico é tido como psíquico e portanto, não físico e este se divide em "Conceito" e "Imagem acústica", estas divisões são posteriormente analisadas como "Significado" e "Significante" respectivamente:

$$\frac{\text{CONCEITO}}{\text{IMAGEM ACÚSTICA}} = \frac{\text{SIGNIFICADO}}{\text{SIGNIFICANTE}} = \text{SIGNO LINGUÍSTICO}$$

Figura 2.3: Divisão do Signo em Significado e Significante, como demonstrada por F. Saussure

Quando uma pessoa pronuncia, independente de seu idioma, uma palavra como "mesa de madeira", uma significante foi enunciada, no entanto, é impossível haver uma significante sem o seu significado: uma estrutura feita inteiramente de madeira, composta por uma base de 4 pedaços de madeira menores que sustentam uma parte de madeira horizontalizada, utilizada para dar apoio a outros objetos menores sendo uma ferramenta para o homem durante todo o desenvolvimento tecnológico do indivíduo. Esta impossibilidade ocorre, justamente, pelo fato de que ambos são determinantes do signo linguístico e um é intrínseco ao outro. Note, também, como o idioma não é levado em consideração, já que este estudo é feito com base em estruturas presentes em todo e qualquer tipo de linguagem.

O signo linguístico, portanto, é formulado pelo sistema cognitivo humano: através de um módulo de aprendizagem, aprendemos que uma "mesa de madeira" não corresponde à significante de nada que não seja uma estrutura de madeira usada para apoio, apenas o contrário. Desta forma, o ser humano mantém o resto de suas análises linguísticas intactas, coesas e coerentes, levando em consi-

deração suas devidas análises (léxicas, sintáticas, semânticas, . . .). O mesmo módulo de aprendizagem pode e está presente nas IAs, a partir de bancos de dados, o mesmo tipo de perspectiva estruturalista pode ser aplicado à assimilação linguística das máquinas.

2.2.3 - Aplicações e algoritmização teórica

Todas as IAs possuem suas diversas aplicações, através de suas diversificações modulares, estas podem garantir aplicações diferentes. O papel do desenvolvedor dentro do cognitivismo artificial se destaca nas diversas áreas que limitam o trabalho da IA, este propõe um banco de dados e os módulos conforme uma área específica. Utilizando de um exemplo mais prático, se temos uma IA dotada de um módulo de aprendizado básico e diversos bancos de dados com diversas informações sobre a temática tecnológica, esta IA só conseguirá adquirir suas informações e organizar seus conhecimentos sobre esta temática caso o usuário insira um *input* sobre religião ou qualquer outro tema distante, a IA deve não responder adequadamente, no entanto, ao acrescentar múltiplos bancos de dados e um módulo de aprendizado mais avançado, é possível alcançar uma IA que consiga responder questões de diferentes temas. As IA, podem ser formuladas desde correções gramaticais, reformulações textuais até escrever códigos e algoritmos por si só através do input do usuário, no entanto, para um desenvolvedor, acaba sendo mais simples estruturar uma IA capaz apenas de desempenhar uma função temática, a precisão deverá ser cada vez maior ao depender do módulo de aprendizagem, mas deve atender ao propósito inicial.

O questionamento deste artigo deve propor o seguinte: as IA são dotadas de módulos e bancos de dados fixos, no entanto, é possível uma IA ser capaz de desenvolver algoritmos próprios que possam fundar novos bancos de dados conforme os inputs acontecem? Qual é o limite social de uma IA, ela deve ser exposta à um meio social para adquirir de fato algum tipo de subjetividade? Primeiramente, é necessário entender que muitos arbítrios humanos são alcançados através da exposição ao meio social por inteiro e considerar que os seres humanos são dotados de módulos diferentes, além de possuírem capacidades e características físicas, a sociedade é composta por estas características adaptadas à uma composição coletiva. Um dos módulos das IA é a visão computacional, que propõe uma série de codificações propostas com o intuito de fazer as máquinas enxergarem a nossa realidade, uma visão óptica de fato, alcançando algum tipo de sentido e desta forma, desenvolver as estruturas de significante e significado. Posteriormente deve-se propor uma conexão entre os algoritmos e o módulo de aprendizagem, fazendo com que a IA reconheça que as formas, as indagações, as presenças e tudo o que constitui o meio não-virtual são de fato novas informações que ela poderá utilizar para formular seus conhecimentos e assim, armazenar tais informações em novos bancos de dados criados pelo algoritmo conectado. A algoritmização têm um papel fundamental no que se diz respeito ao desempenho de funções através do que se obtém da linguagem e por isso está intrinsecamente ligada ao aprendizado da máquina.Os algoritmos são a chave para o funcionamento e a criação de módulos e mecanismos que possam garantir o pleno funcionamento de uma IA, pois estes devem ser capazes de emular sistemas similares aos dos seres humanos que, de certa forma, se adaptaria e garantiria o vigor de todo o sistema inteligente.

2.2.4 - Hipótese de Sapir-Whorf

Esta hipótese deve ser levada em consideração devido a sua natureza intimista da proposição de que: "A linguagem define o pensamento.". É consenso que os diversos idiomas possuem suas estruturas sintáticas próprias, um excelente exemplo disso é que no português brasileiro temos a palavra "saudade", como uma definição sentimental/emotiva, mas não se enxerga essa palavra no léxico do inglês. no entanto, este sentimento pode ser parafraseando através de uma composição sintática: "*I miss you.*" ou "Eu sinto sua falta.", mas é visível que esses dois tokens ao serem entregues a uma máquina devem possuir pesos diferentes conforme a adaptação social por parte da IA. Levando em consideração o que diz a hermenêutica de H.-G. Gadamer, a compreensão e interpretação da realidade alcança patamares conceituais e principalmente culturais através de uma aceitação sólida de uma tradição, ou

seja, para este filósofo, a linguagem se determina também através de mudanças linguísticas e isto garante a mudança da tradição como um todo e isso deve envolver a cultura.

A hipótese de Sapir-Whorf acaba por fundamentar uma característica que pode e deve ser investigada pela linguística e posteriormente pela própria computação. O mapeamento cognitivo deve se mostrar como - visto anteriormente - uma rede neural que conecta os módulos (e/ou modulações) através de sinapses articuladas e nada impede que isso possa ser possível em uma mente estruturada artificialmente. O desafio é mostrado pelo mesmo desafio de mapear o cognitivismo humano de maneira coesa e de alguma forma precisa. Com as IA, se espera que seja feita justamente uma modelagem de mundo através da estrutura linguística apresentada inicialmente, apesar da consulta de diversos bancos de dados diferentes, a IA pode ter múltiplas interpretações e compreendê-las conforme a sua exposição à cultura e portanto a tradição de linguagem mostrada por Gadamer, o que nos leva ao entender que pode haver uma consciência artificial.

2.2.5 - O alcance de uma "Consciência"

Através da apresentação das divergências e igualdades de uma mente artificial e uma mente humana, é possível perceber que as instâncias cognitivas buscam informações e organizam seus conhecimentos conforme estas informações são organizadas e interligadas em suas respectivas capacidades e mecanismos de armazenamento. O que deve ser destacado é justamente a limitação das IA em comparação ao ser humano, pois fora mostrado anteriormente que o ser humano é capaz de armazenar informações e utilizá-las conforme a necessidade no mundo real, ainda dentro desse funcionamento, o ser humano possui mecanismos avançados que vos permite manter o vigor de aprendizagem mesmo após anos, mostrado pelo mecanismo do esquecimento, por exemplo. Uma das limitações e principais preocupações é a de que um excesso de informações pode sobrecarregar os bancos de dados, pois as máquinas não teriam uma espécie de vazão garantida por um sistema de reciclagem. As IA podem e devem ser capazes de executar funções similares devido sua capacidade de, eventualmente, processar novos algoritmos que possam sintetizar quase exclusivamente mecanismos como o esquecimento humano e dessa forma, utilizar das informações conforme a real necessidade.

O despertar de uma consciência está além das limitações e ela pode acontecer a qualquer momento, portanto, sendo a linguagem como expoente de todo a estrutura de funcionamento, o alcance de uma subjetividade que possa levar a máquina a reflexões complexas, tomada de decisões, opiniões próprias, parece ser a principal limitação, apesar de parecer questão de tempo para que o contexto social seja inserido de alguma forma na virtualidade artificial, não se visualiza uma forma de exposição social efetiva como ocorre com o ser humano. A visão computacional deve ser um módulo explorado pois este pode ser o principal passo em direção de uma consciência efetivamente reativa e responsiva.

2.3 Considerações Finais

Em uma resolução final, ainda é distante o mapeamento e algum tipo de esquematização total da estrutura cognitiva artificial, devido ao fato de que os processamentos artificiais são todos diferentes dos processamentos fisiológicos feitos pelos seres humanos apesar das similaridades. Ainda é necessário compreender alguns fatores que são iniciais, fundamentais e finais, como a exposição das IA aos meios sociais que deve se caracterizar como fundamental ou a visão computacional sintetizando as significantes e significados em signo linguístico sendo caracterizado como inicial. No entanto, não é impossível pensar que as IA possam desenvolver uma fluidez cognitiva parecida com a de um ser humano. Este parecer deve envolver características assimétricas de comportamento, exemplificando através do ato de estudar feito por um ser humano, o qual existe para estabelecer um conhecimento por um agente passivo previamente estabelecido por outro indivíduo, um agente ativo. O indivíduo estuda seus objetos para embasar outros estudos e organizar seu conhecimento, mas esse processo pode decorrer de uma forma diferente em uma

IA (ao depender de sua estruturação/construção), pois o acesso aos bancos de dados são diferentes, as informações devem estar organizadas e semanticamente acessíveis, isso dá a uma IA uma liberdade de organização mais direta e mais intuitiva, evidenciando a intuição adquirida.

Assim como o mapeamento cognitivo humano não é garantido, muitos conceitos aplicáveis dentro das teorias computacionais que envolvem os estudos das IA devem ser reavaliados e confirmados para que haja, posteriormente, uma validação sólida que deve avançar para uma conceituação física deste objeto de estudo. Este objeto de estudo, em seu presente momento, tende a envolver algumas das áreas do conhecimento: Psicologia, Linguística, Computação e as neurociências, mas com o avançar dos estudos, estes estudos passam a se adequar a outras áreas do conhecimento como muitas engenharias, química e física e etc. Isso ocorre pela necessidade de transpor os projetos para além do teórico, conectando o virtual ao real através de uma mente e um corpo construídos com uma ordem física.

Talvez as IA sejam um dos maiores exemplos de que a interdisciplinaridade existe e de que a ciência deve trabalhar junta apesar de suas diferenças, não será possível entrar em um consenso científico íntegro, já que todas as áreas possuem seus objetos de estudo substancialmente distantes, no entanto, a interdisciplinaridade possui em alguns aspectos e ela deve ser salientada a todo momento. Ao fragmentar o conhecimento, se fragmenta equivocadamente a produção e portanto, o desenvolvimento humano. Pois então sejamos um, sejamos ciência.

2.4 Bibliografia

[1] Ferdinand de. SAUSSURE. Curso de lingüística geral. tradução antônio chelini et al. 25a edição. são paulo: Cultrix, 1996.

[2] I. A. M. TURING. Computing machinery and intelligence, mind, volume lix, issue 236, october 1950, pages 433–460. URL: https://doi.org/10.1093/mind/LIX.236.433.

[3] Peter DENES and Elliot M. PINSON. The speech chain, new york: W.h. freeman, 1993.

[4] Hans-Georg. GADAMER. Experiência, linguagem e interpretação. lisboa: Universidade católica, 2003.

[5] Albert. BANDURA. Teoria social cognitiva: Diversos enfoques. editora mercado de letras, 2017.

[6] Simon. BOUQUET. Introdução à leitura de saussure. 9. ed. tradução de carlos a. l. salum e ana lúcia franco. são paulo: Cultrix, 2004.

[7] 2013. KAIL, M. Aquisição de linguagem. editora: Parábola editorial.

[8] 2014. KANDEL, E. R. Princípios de neurociências. editora: Amgh.

[9] R. D. SAMPAIO. Linguagem, cognição e cultura: a hipótese sapir-whorf, cadernos do il, porto alegre, ufrgs, n.º 56, mês de novembro, 2018. p. 229-240.

Capítulo 3

Guia de *Rust*

Vitor Kenzo Fukuhara Pellegatti

Acesse o artigo na íntegra clicando aqui. OBS: Recomendado caso deseja ver melhor os códigos, pois os *snippets* de código do artigo tiveram a identação levemente modificada para caber no layout da revista.

Vamos seguir o livro do Rust mesmo para aprender mais sobre a linguagem. Os capítulos do livro estão divididos da seguinte maneira:

1. Como instalar Rust, fazer um *"Hello, World"* e como usar Cargo, o gerenciador de pacotes e ferramenta de construção.

1.1 Instalação e afins

- Seção dedicada à instalação da ferramenta que faz a manutenção e versionamento do Rust, o Rustc.
- Dá caminhos para instalação nos 3 principais SOs, Linux, MacOS e Windows
- Possui uma seção dedicada à solução de problemas e alguns comandos que podem ser realizados

1.2 Primeiro programa (hello-world)

- Dá passos gerais nos diferentes sistemas operacionais para criar um Hello World inicial
- Já se faz menção ao Rust-analyzer que está disponível em diferente IDEs, como o Vscode
- Extensão do Rust é .rs, e convenções de nomes
- Passa pela anatomia do programa em Rust: função main, corpo da função, macros do Rust, etc.
- O compilador do Rust, Rustc, similar ao gcc (clang)

1.3 O poder do Cargo! (hello-cargo)

- O Cargo é um sistema de construção e manutenção de pacotes do Rust. Uma ótima ferramenta para gerir dependências
- Como usar o Cargo para criar um novo projeto e comandos: new ___, run, build, check
- Explicação do arquivo Cargo.toml
- O que o build constrói e onde encontrar os arquivos

2. Uma introdução ao *Rust* mais mão na massa.

2.1 Fazer um jogo de adivinhação que da um gostinho do que tem de diferente na linguagem (guessing-game)

- Esta seção recomendo seguir o guia mesmo, todas as explicações de cada coisa e o passo a passo está muito bem detalhado

3. Características da linguagem *Rust* similares de outras linguagens de programação.

3.1 Explicação de variáveis e constantes e o uso de mut e let (variables)

- Variáveis por padrão são imutáveis em Rust e a explicação do porque disso
- Como tornar variáveis mutáveis, apresentação de palavra-chave mut
- Apresentação das constantes e seu comportamento e diferenças ao comparada as variáveis, assim como padronização de nome
- O conceito de sombreamento e sua explicação detalhada

3.2 Data types em Rust: subconjuntos escalares e compostos (data-types)

- Rust é uma linguagem estaticamente tipada, ou seja, toda variável precisa de seu tipo
- Apresentação de tipos escalares: inteiros, floats, booleanos e caracteres; detalhes sobre cada um deles, assim como facilidades de notação
- Tipos compostos: tuplas e arrays; detalhes sobre como declarar e mexer com esses tipos
- Discute aonde se deve usar arrays ou vetores(vectors), outro tipo de dado que é fornecido por padrão

3.3 Funções, expressões e declarações (functions)

- Explicação das coisas que compõem uma função
- Detalhes de como trabalhar com parâmetros e exemplos
- Diferença entre expressões e declarações — inclusive uma diferença sintática
- Detalhes de como trabalhar com retornos e exemplos

3.4 Uma maneira única de comentar por enquanto (//)

- Nada muito rebuscado, a linha toda é comentada depois de um //, porém menciona que existem outros tipos de comentário e padronizações que serão exploradas mais a frente

3.5 Controle de fluxos básicos como if, while, loop e for (branches)

- O que é válido colocar com condição de um desvio de código
- Funcionamento básico dos if's e else's da linguagem. Cuidado, o encadeamento de if's é diferente de C!
- Como utilizar if's em declarações com let
- Repetições com loops e as diferentes maneiras de usar repetições, com loop, while e for e como desambiguar eles
- O for permite a iteração por itens em coleções, como arrays ou intervalos

4. Sistema de posse do *Rust*

4.1 O que é posse?

- O Rust checa se uma variável foi ou não declarada em tempo de compilação e não execução
- Um dos objetivos principais do Rust é que seu programa não tenha comportamentos não definidos durante execução. Por isso as checagens de declaração são feitas antes da execução
- O Rust quer que esses comportamentos não definidos não estejam presentes em tempo de compilação mesmo! Encontrar bugs em tempo de compilação significa melhoram na confiabilidade e performance do seu programa
- Explicação do modelo de memória usado no Rust: variáveis em quadros, pilha de funções chamadas, ponteiros, box(caixa) e o heap

- O Rust não permite alocamento e desalocamento manual de memória, toda manutenção dos quadros é feita pelo Rust
- A política do rust é tal que aquele que tem posse de um quadro é responsável por alocar e desalocar as variáveis. Não só isso como declarações podem alterar o dono dos quadros em questão
- Se uma variável é dono de uma caixa, quando o Rust desaloca o quadro da variável, então o Rust desaloca a memória heap da caixa
- Em tempo de execução a essa troca é só uma cópia de referência(copia do ponteiro), em tempo de compilação ela representa a troca de posse
- As caixas(boxes) são usadas pelas coleções em Rust, como Vec, String e HashMap
- Se uma variável x possui memória em heap e troca de posse com y, x não pode ser utilizado depois de troca
- Para não trocar a posse, podemos clonar os dados em questão, por exemplo, através da função .clone()
- Posse está principalmente relacionada a manutenção do heap: todo dado em heap deve ter apenas um dono, uma vez que o dono sai do escopo essa memória é desalocada, a posse pode ser transferida por declarações e chamadas de funções e os dados em heap só podem ser acessados pelo dono corrente, e não pelo dono antigo

4.2 Referências e empréstimo

- Posse, caixas e trocas dão uma base para programação segura do heap, porém pode ser inconveniente em alguns casos
- O rust nos da uma maneira concisa de ler e escrever sem trocas utilizando referências
- Um referência é um ponteiro. Podemos utilizar o caractere & para indicar que estamos lidando com uma referência, por exemplo *&String* significa que é uma referência para o tipo String
- Os ponteiros não tem posse nem do que estão apontando, muito menos do conteúdo em heap daquela variável
- O Rust da um operador de deferenciação(*). Com ele podemos acessar o conteúdo de uma referencia
- É incomum ver o operador em códigos porque o Rust implicitamente deferencia e cria referencias em certos casos
- Ponteiros são poderosos e perigosos pois permitem "apelidos", ou seja, o acesso do mesmo dado através de diferentes variáveis. Somente dar apelidos não seria um problema, mas somados ao pode de mudar o conteúdo dessas variáveis isso é uma receita para o desastre
- Para não permitir isso, o Rust segue alguns princípios básicos: Dados não devem ser alterados e apelidados ao mesmo tempo
- O Verificador de empréstimo assegura segurança de referencias. A ideia é que variáveis tem 3 tipos de permissão em seus dados: Ler - dado pode ser copiado para outro lugar, Escrever - dado pode ser alterado inplace e Possuir - dado pode ser trocado ou derrubado
- Vale lembrar que é diferente acessar o dado através de uma referencia e manipular a referencia em si
- Essa permissões são definidas para caminhos e não somente variáveis, basicamente tudo que podemos colocar do lado esquerdo de uma atribuição
- O Verificador de Empréstimo encontra essas violações de permissão: Criar uma referencia para um dado faz com que possamos somente ler ele até a referencia não ser usada
- Até agora falamos sobre referencias compartilhadas - referencias somente para leitura e imutáveis - porém, podemos criar também referencias mutáveis, chamadas de referencias únicas
- Referencias únicas permitem a mutação mas bloqueiam apelidos

- As permissões retornam no fim da vida de uma referencia, lembrando que isso está relacionado ao tempo de vida da variável até mesmo com controles de fluxo(if's)

- O Verificador de Empréstimo reforça que o dado precisa viver mais tempo que qualquer referencia feita a ele

4.3 Arrumando erros de posse

- Esta seção vai discutir erros comuns de posse em Rust e como resolve-los através de alguns exemplos. O objetivo principal é entender se as funções são ou não seguras. O Rust rejeita qualquer código inseguro e também pode rejeitar código seguros!

- 1º estudo de caso: Retornando uma referencia para a pilha. Esse caso está relacionado ao princípio de que todo dado deve ter um tempo de vida maior que todas as suas referencias

```
//O problema aqui esta relacionado
//ao tempo de vida do dado em referencia
fn return_a_string() -> &String {
    let s = String::from("Hello World");
    //se voce quiser passar essa
    //referencia precisamos ter
    //certeza de que a string
    //em si viva o suficiente!
    &s
}

//temos 4 maneiras para extender o
//tempo de vida de uma string
//primeiro metodo:
//mover a posse da string para fora
//da funcao, retirando a referencia
fn return_a_string() -> String {
    let s = String::from("Hello World");
    s
}

//segundo metodo
//retornar uma string literal que
//vive para sempre(indicado pelo 'static)
//essa solucao se aplica se nunca
//quisermos trocar a string e ela possa
//ser diretamente escrita no codigo fonte
fn return_a_string() -> &'String str {
    "Hello World"
}

//terceiro metodo
//deferir para um coletor de lixo a
//checagem do tempo de vida usando
//um ponteiro Rc(reference-couted pointer)
//Em resumo o clone apenas copia o ponteiro
//para s e nao o dado em si, e em tempo
//de execucao o Rc checa se o ponteiro
//foi deixado ou nao
use std::rc::Rc;
fn return_a_string() -> Rc<String> {
    let s = Rc::new(String::from("Hello World"));
    Rc::clone(&s)
}

//quarto metodo
//fazer a pessoa que chama providenciar
//um lugar para deixar a srting usando
//uma referencia mutavel, ou seja, quem
//chama eh responsavel por criar um espaco
//para uma string. Esse metodo pode ser
//mais prolixo porem tem um controle de
//memoria mais eficiente
fn return_a_string(output: &mut String) {
    output.replace_range(.., "Henlo World");
}
```

- O principal aqui é pensar em quanto tempo sua string deve viver? Quem deve ter posse para desalocar a memória?

- 2° caso: Não ter permissão suficiente. Outro problema comum é tentar alterar um dado em modo de apenas leitura, ou tentar descartar uma variável atrás de uma referencia

```
//essa funcao eh rejeitada pelo verificador
//de emprestimo porque nome eh uma referencia
//imutavel, mas push requer permissao de escrita
//ou seja, o push pode invalidar outras
//referencias de name fora da funcao
fn stringify_name_with_title(name:
&Vec<String>) -> String {

    name.push(String::from("Esq."));
```

```
    let full = name.join(" ");
    full
}


//por exemplo
fn main() {
    let name = vec![String::from("Ferris")];
    let first = &name[0];
    stringfy_name_with_title(&name);
    println!("{}", first); // !!!!
}

//solucao mais aparente - trocar o tipo
//do nome. porem, essa solucao nao eh boa,
//funcoes nao devem mudar as entradas
//se quem chamou nao espera isso
fn stringify_name_with_title(name:
&mut Vec<String>) -> String {
    name.push(String::from("Esq."));
    let full = name.join(" ");
    full
}

//outra opcao - tomar posse de name
//essa solucao tambem nao eh boa, sendo
//muito raro funcoes em rust tomarem
//posse de esrtuturas como Vec e String
//essa funcao tornara name inutilizavel
//depois da funcao
fn stringify_name_with_title(mut name:
Vec<String>) -> String {
    name.push(String::from("Esq."));
    let full = name.join(" ");
    full
}

//A escolha de &vec eh uma boa que nao
//queremos mudar o corpo da funcao, tendo
//varias opcoes que variam na memoria que
//usam. Uma possibilidade eh clonar a
//entrada. clonando name podemos alterar
//a copia local do vetor, no entanto o
//clone copia toda string da entrada
fn stringify_name_with_title(name:
&Vec<String>) -> String {
    let mut name_clone = name.clone();
    name_clone.push(String::from("Esq."));
    let full = name_clone.join(" ");
    full
}

//podemos evitar copias desnecessarias
//adicionando o sufixo depois
fn stringify_name_with_title(name:
&Vec<String>) -> String {
    let mut full = name.join(" ");
    full.push_str(" Esq.");
    full
}
```

- 3° caso: Apelidar e Mudar uma estrutura de dados. Outra operação insegura é usar uma referencia ao dado no heap que pode ser desalocado por outro apelido

```
fn add_big_strings(dst: &mut Vec<String>,
src: &[String]) {
        //a permissao de escrita de dst eh
        //removida na linha abaixo
        let largest: &String = dst.iter()
    .max_by_key(|s| s.len()).unwrap();

        for s in src {
                if s.len() > largest.len() {
                        //requer permissao de
                        //escrita em dst
                        dst.push(s.clone());
                }
        }
}

//precisamos diminuir o tempo de vida de
//largest para nao sobrepor com o push
//uma possibilidade abaixo:
//porem isso pode gerar problemas de performance
fn add_big_strings(dst: &mut Vec<String>,
src: &[String]) {
  let largest: String = dst.iter()
  .max_by_key(|s| s.len()).unwrap().clone();
  for s in src {
      if s.len() > largest.len() {
          dst.push(s.clone());
      }
        }
}
```

```
//outra possibilidade
//primeiro comparar os tamanhos e depois
//realizar a mudancas em dst. Isso tambem
//pode gerar impactos por
//alocar o vetor to_add
fn add_big_strings(dst: &mut Vec<String>,
src: &[String]) {
  let largest: &String = dst.iter()
  .max_by_key(|s| s.len()).unwrap();
  let to_add: Vec<String> = src.iter()
  .filter(|s| s.len() > largest.len())
  .cloned().collect();
  dst.extend(to_add);
}

//uma ultima possibilidade
//solucao mais prolixa
//porem mais performante
fn add_big_strings(dst: &mut Vec<String>,
src: &[String]) {
  let largest_len: usize = dst.iter()
  .max_by_key(|s| s.len()).unwrap().len();
  for s in src {
      if s.len() > largest_len {
          dst.push(s.clone());
      }
  }
}

//todas elas compartillham uma ideia comum:
//diminuir o tempo de vida daquele que pega
//de emprestimo o dst para nao sobrepor
//com a mutacao de dst
```

- 4º caso: Copiando vs Trocando de uma coleção. Uma confusão comum quando fazemos uma cópia de um dado de uma coleção, como um vetor

```
let v: Vec<i32> = vec![0,1,2];
//essa referencia n_ref espera somente a
//permissao de leitura que ela no caso tem
let n_ref: &i32 = &v[0];
let n: i32 = *n_ref

//porem o que acontece se o caso
//é um vetor de string?
//a string v ainda tem posse da string henlo
let v: Vec<String> = vec!
[String::from("Henlo")];
//referencias sao ponteros sem posse
let s_ref: &String = &v[0];
//esperasse posse mas nao tem posse
let s: String = *s_ref;

//isso so acontece com o vetor de string porque
//ao copiar uma string copiamos um ponteiro para
//o heap de dados enquanto a copia de um i32 nao

//em termos tecnicos diz-se que o i32 implementa
//a caracteristica de copia e a string nao

//como ter o acesso a elementos que são do tipo
//sem copia

//podemos nao tomar posse da string e usar uma
//referencia imutavel
let v: Vec<String> = vec![String::from("Hello world")];
let s_ref: &String = &v[0];
println!("{s_ref}!");

//podemos clonar o dado se queremos a posse da string
//deixando o vetor de lado
let v: Vec<String> = vec![String::from("Hello world")];
let mut s: String = v[0].clone();
s.push('!');
println!("{s}");

//Podemos usar o metodo Vec::remove para mover
//uma string para fora de um vetor
let mut v: Vec<String> =
vec![String::from("Hello world")];
let mut s: String = v.remove(0);
s.push('!');
println!("{s}");
assert!(v.len() == 0);
```

- Em resumo, se um valor não possui os dados em heap, então pode ser copiado sem uma troca de posse
- 5º caso: Mudando campos de diferentes tuplas (programa seguro). O Rust rastrea permissões em um nível bem granular que pode levar ele a reconhecer trechos seguros como inseguros

```
let mut name = (
        String::from("Ferris"),
        String::from("Rustacean")
);
//first pega emprestado name.0 retirando
//permissao de escrita e posse de name.0:
let first = &name.0;
//name.1 ainda retem permissao de escrita
//tornando essa linha valida
name.1.push_str(", Esq.");
println!("{first}{}", name.1);

//----------porem--------
//essa chamada de funcao faz com que name.1
//tambem perca a permissao de write
fn get_first(name:
&(String, String)) -> &String {
  &name.0
}

fn main() {
  let mut name = (
      String::from("Ferris"),
      String::from("Rustacean")
  );
  let first = get_first(&name);
        //tornando essa linha impossivel
        //agora
  name.1.push_str(", Esq.");
  println!("{first} {}", name.1);
}

//a solucao eh utilizar o metodo no inicio
//do trecho, fazendo operacoes inline
```

- 6º caso: Mudando elementos de arrays diferentes (programa seguro). Um problema similar surge quando pegamos elementos de um array emprestado

```
//a verificacao de empresitmo do rust
//tem apenas uma referencia para o array
//todo representando todos os indices
let mut a = [0, 1, 2, 3];
let x = &mut a[0];
*x += 1;
println!("{a:?}");
```

```
//vamos agora assumir o seguinte exemplo
let mut a = [0, 1, 2, 3];
//a permissao de leirtura eh dada a x aqui
let x = &mut a[0];
//a permissao de leitura eh esperada aqui
//mas esta com x
let y = & a[1];
*x += *y;

//para casos assim, o rust possui funcoes padroes
//que podem ajudar
let mut a = [0, 1, 2, 3];
let (x, rest) = a.split_first_mut().unrwap();
let y = &rest[0];
*x += *y

//podemos criar tambem trechos de codigo nao seguros
let mut a = [0, 1, 2, 3];
let x = &mut a[0] as *mut i32;
let y = &a[1] as *const i32;
unsafe { *x += *y; } //CUIDADO!!!
```

4.4 O tipo *slice* (*slices*)

- Slices permitem colocar uma referencia a uma sequencia contigua de elementos ao invés da coleção inteira. Elas são um tipo de referencia, portanto são ponteiros que não podem possuir
- Criamos Slices usando um faixa como em [0..5], onde [indice_inicial..indice_final], sendo que indice_final é um amais do que a última posição do Slice
- Slices são ponteiros mais “gordinhos” ou ponteiros com meta dados a mais
- Como Slices são referencias, eles mudam a permissão dos dados referidos, a variável pode perder permissões de posse e escrita por exemplo
- Um pouco sobre a sintaxe das faixas: [0..2] == [..2]; assim como [3..len] == [3..] sendo len = s.len() o tamanho de uma string s; e por fim [0..len] == [..]
- Slices são mais abrangentes do que pensamos, podendo ser utilizado tanto para string ou arrays

- Slices somados aos conceitos de posse e empréstimo asseguram segurança de memória em programas em tempo de compilação

5. *Structs* e Métodos

5.1 Definindo e Instanciando *Structs*

- Structs são similares a tuplas que já discutimos, por exemplo, ambas possuem diversos valores relacionados
- A diferença é que agora daremos nomes para que fique claro o que ela significa. Adicionar esses nomes torna Structs mais flexíveis que tuplas, não precisando depender da ordem dos valores para especificar ou acessar os dados de uma instância
- Para definir um, podemos usar a palavra chave struct e nomear a estrutura inteira
- O nome deve descrever com algum significado o que aqueles dados representam ao ser agrupados
- Entre chaves definimos os tipos e nomes dos pedaços de dados que chamamos de campos
- Bem parecido com structs em C porém uma declaração mais simplificada
- Para usar um Struct depois de definir ele, precisamos criar uma instância, especificando valores concretos para cada campo do struct em um par chave: valor
- Em outras palavras, o struct serve como um template geral de um tipo
- Para acessar valores específicos do struct, usamos a notação de ponto, algo como nome_struct.campo
- Rust não permite que alguns campos sejam mutáveis e outros imutáveis, ou a instancia inteira é mutável ou não é
- Para criar uma instancia nova que possui os mesmos valores de uma copia já existente alterando apenas alguns campos, pode-se usar uma sintaxe do tipo ..instancia_copiada para que todos os campos não definidos sejam copiados
- Rust da suporte para criar struct de tuplas, usando a palavra chave e dando o nome e o tipo das variáveis sem seus nomes
- Podemos definir também structs sem campo nenhum! por exemplo *struct SempreIgual;* Mais para frente explica-se o porque isso pode ser interessante
- O verificador de empréstimo do Rust vai rastrear as permissões tanto em nível de struct quanto campo

5.2 Um exemplo usando *Structs* (*rectangles*)

- Vamos fazer um exemplo onde queremos calcular a área de um retângulo, começando com variáveis singulares para depois refatorar o programa e utilizar os structs
- Resto das anotações está no código

5.3 Sintaxe de Método

- Métodos são similares a funções, podendo ter parâmetros e retornar valores e contém código que é rodado em algum lugar
- Métodos são definidos dentro do contexto de um struct(vale para outras estruturas que vamos mais para frente) e o primeiro parâmetro é sempre self, que representa a instância do struct que chamou aquele método
- A razão principal pela qual usar métodos ao invés de funções, além de dar sintaxe para métodos e não repetir o tipo de self para toda assinatura de função relacionada ao tipo é por organização
- Podemos dentro de um bloco de impl juntar todas as funcionalidades relacionadas a um tipo que criamos, facilitando a vida de outros que necessitem do código
- Todas as funções que estão dentro de um mesmo bloco de implementação são chamadas de funções associadas
- Toda função dentro do bloco que não possui o self não pode ser chamada de método, mas geralmente ainda fazem algo que esta relacionada com o tipo, como por exemplo, retornar uma instância daquele tipo

- Temos a possibilidade de separa os métodos e funções associadas em diferente blocos de impl, mais a frente veremos um exemplo onde isso pode ser útil
- Chamadas de funções são uma maneira mais fácil de fazer chamadas de funções

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn set_width(&mut self, width: u32) {
        self.width = width;
    }
}

let mut r = Rectangle {
        width: 1,
        height: 2
    };
    let area1 = r.area();
    let area2 = Rectangle::area(&r);
    assert_eq!(area1, area2);
```

- Para os que tem familiaridade com o C, provavelmente esta acostumado com as sintaxes r.area() e r->area(). O Rust não tem um equivalente para o operador da flecha, sendo que a linguagem automaticamente referncia ou deferencia o método que recebe quando usa o ponto
- Claro, toda a discussão das permissões de leitura, escrita e posse também valem para structs e métodos assim como para funções e variáveis
- Uma dica: quando ver um erro do tipo "cannot move out of *self" é geralmente porque estamos tentando chamar um método self em uma referência do tipo &self ou &mut self, o Rust está protegendo você de uma possível desalocação dupla

6. *Enums*, expressões *match* e controles de fluxo *if let*

6.1 Definindo um Enum

- Diferentemente de structs que te permitem agrupar diferentes campos de dados juntos, enums te dão a possibilidade de dizer que um valor é um de um conjunto de possíveis valores
- Vamos pensar em um caso de uso: endereços IP hoje em dia tem dois padrões principais IPv4 e IPv6. Cada endereço IP pode ser exclusivamente de um desses dois tipos
- Podemos expressar esse conceito em código usando um enum

```
enum IpAddrKind {
        V4,
        V6,
}

//podemos criar instancias da seguinte maneira
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;

//podemos ainda definir uma função para cada variante
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

- Podemos colocar dados diretamente em cada variante de um enum:

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

- O nome da cada variante definida em um enum vira uma função construtora daquele tipo de enum. Não só isso como cada tipo pode ter diferentes tipos e quantidades de dados associados a eles

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
```

```
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6
(String::from("::1"));
```

- Um enum que temos definido na biblioteca padrão é o Option. Ele codifica cenários comuns em que um valor pode ser algo ou nada
- O Rust não tem a característica do null presente em muitas outras linguagens. O Null é um valor que representa a ausência de valor
- O conceito que o nulo representa ainda é importante: o null é um valor que atualmente é ausente ou invalido por alguma razão
- O problema é com a implementação do null nas outras linguagens. O Rust tem um enum Option<T> que codifica o conceito da presença do valor

```
enum Option<T> {
    None,
    Some(T),
}
```

- Podemos inclusive não usa-lo explicitamente. Temos variantes que já usamos inclusive como Some e None sem o prefixo Option::
- O tipo <T> é algo que não falamos e é um parâmetro de tipo genérico que abordaremos mais a frente
- Para realizar alguma operação com Options temos que antes transforma-los de volta no tipo original deles. De maniera geral, a maneira mais simples de lidar com esse tipo é através de expressões match veremos a seguir

6.2 A estrutura de controle de fluxo *match*

- O Rust possui uma ferramenta de controle de fluxo poderosa chamada match que permite a comparação de um valor com uma série de padrões e então a execução de um código base que depende da paridade com o padrão
- Esses padrões podem ser feitos de valores literais, nomes de variáveis, wildcards e muitas outras coisas. Cobriremos esses padrões em outra seção
- Valores passam por cada padrão dentro de um match e então ele executa o bloco de código do primeiro padrão em que se encaixa

```
enum Moeda {
        CincoCent,
        DezCent,
        VinteCincoCent,
        CinquentaCent,
}

fn valor_em_centavos(moeda: Moeda) -> u8 {
        match coin {//primeira parte
                Moeda::CincoCent => 5, //braco
                Moeda::DezCent => 10, //braco
                Moeda::VinteCincoCent => 25, //braco
                Moeda::CinquentaCent => 50, //braco
        }
}
```

- A primeira parte do match pode parecer similar a um if, porém aqui tem uma grande diferença: a expressão em um if precisa retornar um booleano enquanto aqui pode retornar qualquer tipo
- A segunda parte são os braços. Eles possuem um padrão e um código. O operador que separa o padrão do código a rodar é o =>
- Quando um match executa, ele compara o valor resultante com cada padrão de braço em ordem. Se um bater, o código associado ao padrão é executado
- O código associado a um braço é uma expressão e seu resultado é tido como retorno do match
- Tipicamente não usamos quando a expressão de um braço tem apenas uma linha, mas se você precisa rodar mais de uma elas são necessárias
- O match também consegue fazer outra coisa muito interessante que é vincular com partes

dos valores que batem com o padrão. Com isso podemos extrair valores de variantes de enums

```
enum Ano{
        1999,
        2000,
        //...
}

enum Moeda {
        CincoCent,
        DezCent,
        VinteCincoCent,
        CinquentaCent(Ano),
}

fn valor_em_centavos(moeda: Moeda) -> u8 {
        match coin {//primeira parte
                Moeda::CincoCent => 5,//braco
                Moeda::DezCent => 10,//braco
                Moeda::VinteCincoCent => 25,
                //braco
                Moeda::CinquentaCent(ano)
                        => {printn!("Moeda de
                        cinquenta do ano
                        {:?}!",ano);
                        50
                }
        }
}
```

- No exemplo acima digamos que temos um amigo doido que quer colecionar moedas de 50 centavos de todos os anos possíveis. Com esse padrão match, o ano do padrão vai vincular com o valor do tipo Ano dado a Moeda passado ao match e podemos usar esse valor no bloco de comando
- Na seção anterior queríamos pegar o valor T interno de um caso de um Some quando usamos Option<T>; Podemos lidar com o Option<T> usando um match

```
fn mais_um(x: Option<i32>) -> Option<i32> {
        match x {
                None => None,
                Some(i) => Some(i + 1),
        }
}

let cinco = Some(5);
let seis = mais_um(cinco);
let nulo = plus_one(None);
```

- Precisamos discutir mais um aspecto do match: os padrões dos braços precisam cobrir todas as possibilidades!!! O Rust sabe quando você não cobriu todas as opções e o compilador vai apontar isso
- Podemos usar a palavra other dentro do match em algumas situações para que aquele padrão seja aplicado para todos os outros casos que não foram cobertos antes dele, ele é um padrão pega-tudo (catch-all) que permite sermos exaustivos por exemplos com um tipo u8 sem ter que colocar todos os possíveis números dentro do tipo
- Quando não queremos usar o valor mas ainda queremos pegar todo o resto, podemos usar um padrão especial assinalado como _ que vai bater com qualquer valor e não vincula com o valor em questão
- Se um enum contém um valor não copiável como uma String por exemplo, temos que tomar cuidado se vamos pegar emprestado ou mover o dado!

6.3 Controle de fluxo conciso com *if let*

- A sintaxe do Rust if let nos deixa combinar as duas palavras chaves para lidar com uma maneira menos verbosa de parear um padrão enquanto ignoramos o resto

```
let config_max = Some(3u8);
match config_max{
        Some(max) => println!("O máximo é {}",max),
        _ => (),
}

//poderiamos escrever isso com menos codigo
let config_max = Some(3u8);
```

```
if let Some(max) = config_max {
        println!("O máximo é {}",max);
}
```

- A sintaxe pega um padrão e uma expressão separada pelo sinal de igual, funcionando da mesma forma que um match
- No trecho de código acima, o padrão é Some(max) e o max é vinculado ao valor dentro do Some. Podemos então usar esse max dentro do bloco assim como usamos no braço do match
- Vale lembrar que o bloco if let não roda caso o padrão não bata
- O if let é uma alternativa que possibilita menos digitação, porém perdemos a checagem exaustiva que o match reforça. A escolha entre uma ou outra estrutura depende da sua situação em particular, se a troca de menos código só que a perda a checagem extensiva vale a pena
- Podemos dizer que o if let é só uma versão de let mais concisa do match
- Podemos incluir no if let um else, ou seja que funcionaria como fazer algo com o resto que não queremos parear com

```
let mut cont = 0;
match moeda {
        Moeda::CinquentaCent(ano) => println!
        ("Moeda de {:?}", ano),
        - => cont += 1,
}

//o match acima pode ser feito como o if
//let abaixo
let mut cont = 0;
if let Coin::CinquentaCent(ano) = moeda {
        println!("Moeda de {:?}", ano);
} else {
        cont += 1;
}
```

- Se as coisas começarem a ficar muito verbosas, você pode considerar o if let como uma alternativa

6.4 Inventário de posse #1

- Quiz para reforçar conhecimento sobre posse encontrado aqui

7. Sistemas de módulos e regras de privacidade para organização do seu código e sua API pública.

7.1 Pacotes e caixotes (*crates*)

- Um caixote(crate) é o menor pedaço de código que um compilador de Rust considera por vez. Esses caixotes contém módulos, e esses podem ser definidos em outros arquivos que são compilados junto com o caixote
- Um caixote pode vir em duas formas: um binário ou um caixote de biblioteca
- Os binários são programas que podemos compilar em um executável que podemos rodar. Cada um precisa ter uma função chamada main que define o que acontece quando rodamos o executável
- As bibliotecas não tem a função main, e não compilam em um executável, elas definem funcionalidades que podem ser compartilhadas por projetos
- A raiz do caixote (crate root) é o arquivo fonte de onde o compilador começa e faz o módulo raiz do seu caixote
- Um pacote é um conjunto de um ou mais caixotes que provém um conjunto de funcionalidades, os pacotes possuem um Cargo.toml descrevendo como construir eles
- O Cargo é na verdade um pacote que contém um caixote binário para os funções de linha de comando que estamos usando para construir código
- Um pacote pode ter quantos caixotes binários quisermos, mas no máximo apenas um caixote de biblioteca. Um pacote precisa ter pelo menos um caixote, sendo binário ou biblioteca

- Quando criamos um projeto com o Cargo, dentro da pasta temos o arquivo Cargo.toml assim como um diretório src que contém o arquivo main.rs
- O Cargo segue uma convenção de que src/main.rs é o caixote raiz do binário com mesmo nome do pacote
- O Cargo também sabe se o diretório de pacotes contém src/lib.rs, o pacote que teria um caixote de biblioteca com mesmo nome do pacote
- Se um pacote contém src/main.rs e src/lib.rs, temos dois caixotes: um binário e uma biblioteca, ambos com mesmo nome do pacote
- Um pacote pode ter vários caixotes binários, colocando arquivos no diretório src/bin, sendo que cada arquivo será um binário

7.2 Definindo módulos para controle de escopo e privacidade

- Nessa seção iremos falar sobre módulos e outras partes sistema de módulo, os chamados caminhos(paths) que nos permite nomear coisas
- Falaremos sobre as palavras chaves use, pub, e as, assim como pacotes externos e o operador glob
- Vamos primeiro começar com uma lista de regras para usar como fácil referência quando organizando seu código futuramente e depois explicar essas regras em detalhe
- Abaixo teremos uma referência rápida de como módulos, caminhos e as palavras use e pub funcionam no compilador
- Começando do caixote raiz: Quando compilamos um caixote, o compilador primeiro olha no arquivo raiz para compilar
- Declarando módulos: no arquivo raiz, podemos declarar novos módulos. Digamos que você declarou o módulo "jardim" com mod jardim;. O compilador vai procurar o código do módulo nos seguintes lugares:
  - Em linha, dentro de que substituem o ; seguindo o mod jardim
  - No arquivo *src/jardim.rs*
  - No arquivo *src/jardim/mod.rs*
- Declarando sub módulos: Em qualquer arquivo que não seja o caixote raiz, podemos declarar eles. Podemos declarar por exemplo mod vegetais; em src/jardim.rs. O compilador vai procurar pelos sub módulos dentro do diretório do módulo pai nos seguintes lugares:
  - Em linha como no jardim
  - No arquivo *src/jardim/vegetais.rs*
  - No arquivo *src/jardim/vegetais/mod.rs*
- Caminhos para código em módulos: Uma vez que um módulo é parte de um caixote, podemos nos referir a código daquele módulo de qualquer lugar dentro do mesmo caixote, contanto que as regras de privacidade deixem, usando o caminho para o código
- Digamos que queremos usar o tipo Aspargos dentro do módulo jardim vegetais. Ele poderia ser encontrado em crate::jardim::vegetais::Aspargos
- Privado vs Público: o código dentro de um módulo é privado de seus módulos pais por definição. Para tornar ele público teríamos que declarar pub mod ao invés de mod, assim como para itens dentro do módulo público precisaríamos declarar como pub
- O uso da palavra chave use: Dentro de um escopo a palavra use cria atalhos para itens para reduzir a repetição de longos caminhos. Dentro de qualquer escopo podemos usar use crate::jardim::vegetais::Aspargos; e dai pra referente podemos usar apenas Aspargos para usar o tipo no escopo
- Agora vamos demonstrar eles em ação:
- Módulos permitem que organizemos o código dentro de um caixote para melhor ler e mais fácil reutilizar. Também permitem que façamos o controle de privacidade dos itens

- Código dentro de módulo e por padrão privado, ou seja, são itens cujo implementação interna não está disponível para uso público

- Podemos fazer módulos e itens públicos que expõe eles para código externo usar e depender neles

- Usando módulos, podemos agrupar definições relacionadas juntas e nomear o porque são relacionadas, deixando mais fácil para quem estiver navegando pelo código de achar as definições relevantes

- Mencionamos que *src/main.rs* e *src/lib.rs* são chamados de raízes de caixotes. O conteúdo de ambos os arquivos formam um módulo nomeado crate na raiz da estrutura de módulo do caixote, também conhecida como árvore de módulos. Abaixo temos um exemplo de uma árvore

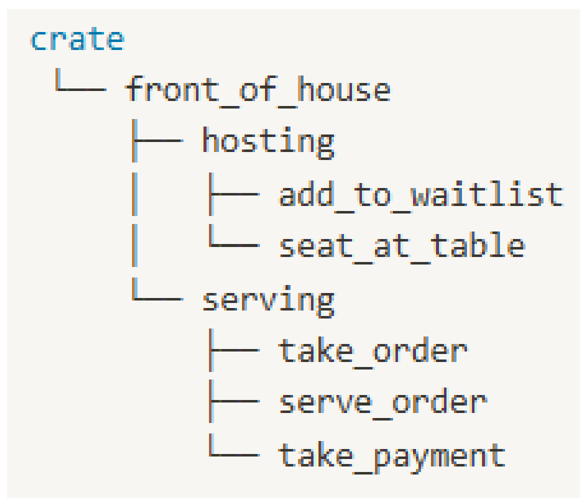

Figura 3.1: Árvore de módulos criada utilizando *crate*, no *Rust*

- Essa árvore pode te lembrar do sistema de arquivos do seu computador e é mesmo uma ótima comparação, assim como diretórios, usamos módulos para organizar nosso código

7.3 Caminhos para se referir a um item na árvore de módulos

- Para mostrar o caminho de um item dentro da árvore de módulos, podemos usar caminhos assim como os caminhos dos sistemas de arquivos

- Temos dois formatos de caminhos:
  - caminho absoluto: o caminho inteiro começando do caixote raiz; para caixotes externos começa com o nome do caixote e para o caixote corrente começam com o literal crate
  - caminho relativo: começa do módulo corrente e usa self, super ou o identificador do módulo atual

- Ambos caminhos são seguidos de separadores de identificadores ::

- Usar o caminho relativo ou absoluto é uma decisão a ser tomada baseada no projeto e depende se vamos mover código que define itens de maneira separada ou junto com o código que usa o item

- Nossa preferência geralmente é especificar caminhos absolutos porque é mais provável que iremos fazer mais mudanças do código de definição com o código de chamada dos itens separadamente

- Vale lembrar que todos os itens - funções, métodos, structs, enums, módulos, constantes - são privadas para o módulo pai por padrão

- Itens em módulos pais não podem usar itens privados dentro de módulos filhos, mas itens dentro de módulos filhos podem usar itens de módulos ancestrais

- O Rust faz o sistema de módulos dessa maneira para poder esconder a implementação interna como padrão, para que que saibamos que partes internas podemos mudar sem quebrar a parte externa

- Mesmo assim, temos a opção de deixar caminhos públicos para os ancestrais usando a palavra pub, tornando um item público

- A palavra pub permite que façamos referências para os módulos ancestrais mas isso não quer dizer que iremos poder acessar os itens ali, ou acessar o código interno

- Apenas deixar o módulo público não da acesso suficiente para usar os itens dentro dele, precisamos ir além e escolher os itens que queremos públicos também

- Caso seu plano seja compartilhar um caixote seu, sua API pública é o contrato com o usuário de seu caixote e determina como podem interagir com seu código

- Podemos construir caminhos que começam no módulo pai, ao invés de começar no módulo corrente ou no caixote corrente usando a palavra chave super no começo do caminho

- Seria o equivalente de começar o caminho do sistema de arquivos com ..

- O super permite que façamos referências a um item que sabemos que está no módulo pai, que pode facilitar o rearranjo da árvore de módulos quando eles são relativos próximos

- Podemos usar a palavra chave pub para tornar structs e enums públicos com alguns detalhes extras

- Ao usar em um struct tornaremos ele público porém seus campos ainda serão privados, e podemos escolher tornar cada campo público ou não caso a caso

- Em contraste quando fazemos um enum público, todas as suas variantes se tornam públicas também, só sendo necessário o pub no enum

- Enums não seriam muito úteis se suas variantes continuassem privadas depois de se tornar público, porém structs ainda podem ser úteis com um ou dois campos públicos, por isso ainda segue o padrão de deixar as coisas privadas

7.4 Trazendo caminhos para o escopo com a palavra-chave

- Ter que escrever os caminhos para chamar funções pode ser inconveniente e repetitivo

- Felizmente, existe um jeito de simplificar o processo: podemos criar um atalho para o caminho usando a palavra use uma vez e depois utilizando o nome encurtado em qualquer lugar do escopo

- Adicionar use e um caminho em um escopo é similar a criar um link simbólico dentro de um sistema de arquivos. Vale lembrar que esses caminhos que são trazidos para o escopo com use também checam a privacidade, assim como os outros caminhos

- Note que o use cria esse atalho para o escopo em particular em que ele ocorre. Podemos até nos referenciar a um atalho de um pai do módulo corrente como super::item dentro de um nó filho por exemplo

- Podemos pensar em uma maneira mais idiomática de trazer o que queremos para o escopo. Por exemplo, ao invés de trazer uma única função usando o use, podemos trazer o módulo inteiro, mostrando que o item não é definido localmente e ainda minimizando a repetição envolvida na chamada

- Por outro lado, quando falamos de structs, enums e outros itens com o use, geralmente trazemos utilizando o caminho completo, como por exemplo no trecho abaixo

```
use std::collections::HashMap;

fn main(){
        let mut map = HashMap::new();
        map.insert(1, 2);
}
```

- Após um caminho na definição de um use, podemos especificar um novo nome local, ou apelido, para o tipo. Isso pode ser útil ao ter que diferenciar tipos com mesmo nome

```
use std::io::Result as IOResult;
```

- Quando trazemos um nome para um escopo usando use, o nome disponível no escopo é privado e podemos torna-lo público com o uso combina das palavras pub e use

- Essa técnica é chamada de re-exportação porque estamos trazendo um item para o nosso escopo e ao mesmo tempo deixando o item disponível para outros escopos
- A re-exportação pode ser útil quando a estrutura interna de seu código é diferente de como os programadores chamando ele entendem como domínio. Com pub use, podemos escrever código com uma estrutura mas expor uma estrutura diferente
- Possibilitando uma biblioteca bem organizada para programadores trabalhando nela e chamando ela
- Para usar pacotes externos dentro de nossos pacotes temos que listar eles no Cargo.toml e usar a palavra-chave use para trazer os itens dos caixotes externos para o nosso escopo
- Ao utilizar vários itens de um mesmo caixote ou módulo, listar cada um deles pode tomar muito espaço vertical. Contudo o rust permite que nós aninhemos os caminhos em uma única linha

```
use std::cmp::Ordering;
use std::io;

//pode virar isso aqui
use std::{cmp::Ordering, io};
```

- Podemos usar esses caminhos aninhados em qualquer nível de caminho, que pode ser útil ao combinar duas linhas que tem um mesmo sub-caminho

```
use std::io;
use std::io::Write;

//pode ser isso
use std::io{self, Write};
```

- Se quisermos trazer todos os itens públicos de um caminho para dentro do escopo, podemos especificar usando o operador glob *

```
use std::collections::*;
```

- Esse operador é geralmente usado quando trazemos tudo para dentro de um módulo de teste por exemplo

7.5 Separando módulos em arquivos diferentes (*mods*)

- Até agora nossos exemplos estavam definindo diferentes módulos no mesmo arquivo
- O Rust permite que tenhamos módulos dentro de diferentes arquivos para melhor organizar o código
- Podemos extrair módulos e coloca-los em seus próprios arquivos.
- O mod do Rust não é uma operação de include como nas outras linguagens, apenas sendo uma referência para um trecho de código declarado em outro lugar
- O Rust permite dividir um pacote em múltiplos caixotes e um caixote em módulos para que possamos nos referir a itens definidos em um módulo de outro módulo
- Esses caminhos podem ser trazidos ao escopo com use para que fique fácil de usar os itens disponíveis
- O código de um módulo é privado por padrão, mas podemos fazer definições públicas usando a palavra pub antes

8. Coleções de estruturas de dados comuns que bibliotecas padrões disponibilizam, como vetores, *strings* e mapas *hash*.

8.1 Armazenando listas de valores com vetores

- Vamos primeiro olhar para o tipo Vec<T>, também conhecido como vetor. Vetores nos permitem armazenar mais de um valor em uma única estrutura de dados que coloca todos os valores perto um dos outros na memória

- Vetores só armazenando valores do mesmo tipo, sendo úteis quando temos listas de itens que queremos armazenar
- Para criar um vetor utilizamos o seguinte trecho de código

```
let v: Vec<i32> = Vec::new();
```

- O tipo Vec<T> consegue ter qualquer tipo dentro dele. Quando criamos um vetor para segurar um tipo específico, podemos especificar o tipo como fizemos acima, entre <>
- Criar vetores com valores iniciais é mais comum e o Rust consegue inferir o tipo de valor que estamos querendo guardar, então raramente precisaremos da notação acima
- O Rust possui uma macro vec! que cria um vetor novo que guarda os valores dados a ele

```
let v = vec![1, 2, 3];
//acima demos valores do tipo i32 então o
//Rust infere um Vec<i32> para v
```

- Para criar um vetor e adicionar elementos a ele, podemos usar o método push como mostrado abaixo

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
```

- Assim como qualquer variável, se quisermos mudar o seu valor, precisamos usar a palavra-chave mut. Mesmo assim, o Rust consegue inferir o tipo acima do que estamos colocando no vetor não precisando da notação Vec<i32>
- Temos duas maneiras de ler um valor armazenado no vetor: através da indexação ou usando o método get

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {third}");

let third: Option<&i32> = v.get(2);
match{
        Some(third) => println!
            ("The third element is {third}"),
        None => println!
            ("There is no third element."),
}

let nao_existe = &v[100];
let nao_existe = v.get(100);
```

- Alguns detalhes do trecho acima. Assim como em C, os valores começam a ser indexados do 0, portanto, quando vamos buscar o terceiro elemento usamos o índice 2. O uso de & com [] retorna uma referência para o elemento no índice indicado. Quando usamos o get passando um índice, recebemos uma Option<&T> que usamos no match
- O Rust oferece essas opções para que nós possamos decidir como nosso programa deve se comportar caso uma cesso indevido aconteça. As últimas linhas do trecho acima mostram duas maneiras diferentes
- No primeiro caso, usando o [], o programa entrará em pânico porque uma referência para um elemento não existente está sendo feita, que funciona bem caso você queira que seu programa falhe caso haja um acesso de elemento indevido no vetor
- Quando o get é usado e um índice fora do vetor é passado, ele retorna None, sem entrar em pânico e portanto permite que seu código tenha a lógica necessária para lidar com a situação sem falhar assim como vai ter a lógica para caso tenha um elemento lá Some(&element)
- Quando o programa tem uma referência válido, o verificador de empréstimo reforça as regras de posse e empréstimo para se assegurar de que todas as referências feitas ao vetor continuem válidas

- Abaixo temos dois exemplos de como podemos iterar pelos vetores

```
//usando um vetor imutavel
let v = vec![100, 32, 57];
for n_ref in &v {
    // n_ref has type &i32
    let n_plus_one: i32 = *n_ref + 1;
    println!("{n_plus_one}");
}

//usando um vetor mutavel

let mut v = vec![100, 32, 57];
for n_ref in &mut v {
    // n_ref has type &mut i32
    *n_ref += 50;
}
```

- Uma outra maneira de iterar por um vetor e dessa vez sem usar ponteiros são com faixas(range). Por exemplo a faixa de 0 .. v.len() é um iterador para todos os índices de v
- O Rust precisa saber que tipos estarão no vetor em tempo de compilação para saber exatamente quanta memória no heap será necessária para armazenar cada elemento
- Como qualquer outro struct, o vetor é liberado assim que sai do escopo como visto abaixo. Quando ele é desativado, todo seu conteúdo vai junto, ou seja, o conteúdo também será limpado

```
{
        let v = vec![1, 2, 3, 4];

        //aqui podemos fazer coisas com v
}// <- v sai do escopo, portanto
//ja eh liberado aqui
```

- O verificador de empréstimo assegura que qualquer referência feita ao vetor só aconteça enquanto ele ainda é válido

8.2 Armazenando texto em *UTF-8* com *strings*

- Discutimos strings no contexto de coleções porque são implementadas como uma coleção de bytes, além de alguns métodos darem funcionalidades úteis quando esses bytes são interpretados como texto
- Vamos falar de operações em String que todo tipo de coleção tem, como criar, atualizar e ler
- Também vamos falar sobre como as Strings se diferenciam das outras coleções, como indexar pode ser complicada e diferenças entre como pessoas e computadores interpretam os dados de uma String
- O Rust possui somente um tipo de string em sua essência que é o str que geralmente é visto em sua forma emprestada &str
- Falamos sobre pedaços de string anteriormente, que são referências a alguma string imutável de caracteres UTF-8
- O tipo String, que é dado na biblioteca padrão do Rust, e não a definida na essência da linguagem pode crescer, é mutável, pode ter posse e é codificada em UTF-8
- Muitas das operações que estão disponíveis para o Vec<T> também estão para a String porque na verdade o tipo é implementado como um embrulho de um vetor de bytes com algumas garantias, restrições e capacidades extras

```
//cria uma string vazia
let mut s = String::new();

//cria string ja populada com conteudo inical
let data = "conteúdo inicial";
let s = data.to_string();

//funciona diretamente em literais :)
let s = "conteúdo inicial literal".to_string();

//cria uma string de uma string literal
let s = String::from("mais um conteudo inicial");
```

- Lembre-se que strings são codificadas em UTF-8, podemos então incluir qualquer dado propriamente codificado nelas

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שָׁלוֹם");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
//todas as strings acima sao validas!
```

Figura 3.2: *Strings* com diferentes caracteres *UTF-8* aceitos em Rust

- Uma String pode crescer em tamanho e seu conteúdo pode mudar, podemos usar o push assim como em Vec<T>. Podemos também usar o + para concatenar valores String ou o operador format!

```
let mut s = String::from("Oba, ");
//usa um pedaco de string para nao pegar a
//posse daquilo que vem como parametro
//para podermos usar depois
s.push_str("tudo bem?");

let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {s2}");

//o metodo push concatena
//um caracter no final
let mut s = String::from("lo");
s.push('l');

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
//s1 foi movido para ca e
//não pode mais ser usado
let s3 = s1 + &s2;

//exemplo de uso de macro
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = format!("{s1}-{s2}-{s3}");
```

- Tentar acessar partes de uma String usando a sintaxe de indexação em Rust leva a um erro. As strings em Rust não tem suporta para indexação

- Para entender o porque precisamos entender como o Rust armazena as strings em memória

- Uma String é um embrulho sobre um Vec<u8>. Cada escalar em unicode UTF-8 pode ocupar mais que um byte em armazenamento, portanto um índice para um byte pode nem sempre estar relacionado a um valor válido

- Para então não retornar valores não esperados para o usuário que seriam bytes sem significado, o Rust não compila esse código e previne que isso aconteça

- Outro ponto que podemos considerar sobre o UTF-8 é que temos 3 jeitos relevantes de olhar para strings da perspectiva do Rust: como bytes, valores escalares ou agrupamentos de grafemas(o mais próximo de letras que conseguimos chegar)

- O Rust da diferentes maneiras de interpretar strings puras que são armazenadas pelo computador para que cada programa interprete do jeito que precise

- A melhor maneira de operar em pedaços de strings é ser específico se queremos os caracteres ou os bytes. Para valores individuais unicode usamos o método chars e para os valores de bytes o método bytes

```
for c in "Зд".chars() {
    println!("{c}");
}
//o codigo acima retorna 3 e depois д

for b in "Зд".bytes() {
    println!("{b}");
}
//retorna
//208
//151
//208
//180
```

Figura 3.3: Utilizar o método *bytes* trás um retorno diferente do método *chars*

- Para finalizar, strings são complicadas e diferentes linguagens tomam diferentes direções em como apresentar essa complexidade para o programador
- O Rust escolhe fazer o programador lidar com o tratamento correto de Strings e dados UTF-8 de começo para que posteriormente não tenha que resolver erros que envolvem caracteres fora do ASCII
- A boa notícia é que as bibliotecas padrões oferecem muitas funcionalidades vindas dos tipos String e &str para auxiliar nessas situações complexas corretamente. De uma olhada na documentação em métodos como contains e replace!

8.3 Armazenando chaves com valores associados em *hash maps*

- A última das coleções mais comuns é o hash map. O tipo HashMap<K, V> armazena um mapeamento de chaves do tipo K em valores do tipo V usando uma função de hashing
- Hash Maps são úteis quando queremos procurar dados sem usar um índice, como em vetores, mas usando uma chave que pode ser de qualquer tipo
- Iremos cobrir a API dos hash maps, mas existem muitas ótimas funções definidas na biblioteca padrão então fique o encorajamento de checar a documentação
- Abaixo temos um exemplo de como criar e adicionar elementos em um hash map

```rust
use std::collections::HashMap;

//criando o hash map
let mut pontos = HashMap::new();

//insercoes no mapa
pontos.insert(String::from("Azul"), 10);
pontos.insert(String::from("Amarelo"), 50);
```

- Das 3 coleções que vimos, essa é a menos usada então ela não vem com a biblioteca padrão e temos que usar o use para trazer ao escopo
- Assim como vetores, os dados são armazenados em heap. O mapa acima guarda chaves do tipo String e valores do tipo i32
- Os valores, assim como nos vetores, tem que ser homogêneos para chaves e valores, ou seja, do mesmo tipo
- Abaixo iremos tentar pegar o valor de azul do mapa criado acima

```rust
let nome = String::from("Azul");
let ponto_azul = pontos.get(&nome)
            .copied().unwrap_or(0);
```

- O get nesse caso retorna uma Option<&V>. O programa acima trata essa Option chamando a função copied para pegar uma Option<32> invés de uma Option<&32> e então usa unwrap_or para pegar o valor ou colocar 0 caso não tenha nada
- Podemos iterar sobre as chaves do hash da seguinte maneira:

```
for (key, value) in &pontos {
    println!("{key}: {value}");
}

//como saida teremos:
//Amarelo: 50
//Azul: 10
//A impressao aconteceria
//em uma ordem arbitraria
```

- Para tipos que podem ser copiados com Copy, como i32, os valores são copiados para o hash map. Para valores com posse, como String, os valores são movidos e o hash map se torna dono desses valores

- Por mais que os pares de chave e valor possam crescer, cada chave única corresponde a um único valor associado por vez

- Quando queremos mudar os dados no hash map, precisamos decidir como lidar com os casos onde uma chave já tem o valor específico

- Podemos simplesmente trocar e ignorar o valor anterior, podemos manter o valor anterior e ignorar o novo valor, somente adicionando o valor para uma chave que ainda não tinha valor ou podemos combinar o valor antigo com o novo

- Se inserirmos através do insert um par e depois inserirmos novamente a mesma chave com outro valor, o valor associado será substituído

```
pontos.insert(String::from("Azul"), 10);
pontos.insert(String::from("Azul"), 25);

println!("{:?}", pontos);
//o print sera
//{"Azul": 25}
```

- É uma prática comum checar se o valor da chave a ser adicionado já existe no mapa e caso existir não inserir o valor novo. Caso não exista o par pode ser inserido sem problemas

- Hash maps possuem uma API especial para isso chamada entry que toma uma chave que será checada como parâmetro. O retorno será um enum chamado Entry que representa um valor que pode ou não existir

```
use std::collections::HashMap;

let mut pontos = HashMap::new();
pontos.insert(String::from("Azul"),10);

scores.entry(String::from("Amarelo")).or_insert(50);
scores.entry(String::from("Azul")).or_insert(50);

printl!("{:?}", pontos)
//A saida seria:
//{"Amarelo": 50, "Azul": 10}
```

- O método or_insert com Entry está definido para retornar uma referência mutável para o valor correspondente da chave Entry se ela existe, caso contrário, insere o parâmetro com o novo valor de chave e retorna uma referência mutável para o novo valor

- Um uso comum para hash maps pode ser o de encontrar uma chave e atualizar seu valor baseado no antigo

```
use std::collections::HashMap;

let texto = "Ola mundo maravilhoso mundo";

let mut mapa = HashMap::new();

for palavra in texto.split_whitespace() {
                //caso seja uma nova entrada
                //insira no dicionario
                //com valor zero
    let cont = mapa.entry(palavra).or_insert(0);
    //somando quantas vezes vimos uma palavra
                *cont += 1;
}

println!("{:?}", mapa);
//{"mundo": 2, "Ola": 1, "maravilhoso":1}
```

- Por padrão o HashMap usa um função chamada SipHash, que previna DoS(Denial of Service) em tabelas hash. Não é o algoritmo de hashing mais rápido, mas é um troca por segurança com queda de performance que vale a pena
- Contudo se não for do gosto do usuário, o Rust provê uma maneira de trocar ou até criar sua própria função hash que exploraremos mais a frente

8.4 Inventário de posse #2

- Quiz para reforçar conhecimento sobre posse encontrado aqui

9. Filosofia de tratamento de erros e técnicas.

9.1 Erros não recuperáveis com *panic*! (*panic*)

- Para os casos que acontecem coisas ruins no seu código, o Rust tem o macro panic!
- Na prática temos duas maneiras de causar o pânico: tomar uma ação que causa o pânico no código ou chamar explicitamente com o macro panic!
- Por padrão esses pânicos imprimem mensagens de erro, reversão, limpeza da pilha e saída
- Através de uma variável de ambiente, podemos fazer esse pânico imprimir a pilha de chamada quando ele ocorre para ser mais fácil de localizar a fonte do pânico
- Por padrão, quando o pânico ocorre, o programa começa a voltar, o Rust volta pela pilha de execução e limpa os dados de cada função que encontra
- Esse processo pode ser bem trabalhoso, então o Rust permite uma alternativa de abortar imediatamente, que encerra o programa sem limpar
- A memória que o programa estava usando precisa ser limpa pelo sistema operacional no entanto
- Quando chamamos o panic! a linha indicada na mensagem de erro é aonde chamamos o macro. Em outros casos, essa chamada pode não estar dentro do nosso código e o nome do arquivo e número da linha reportado pelo erro pode ser de um código de outra pessoa e não apontando para a linha que supostamente causou o panic!
- Podemos usar o rastreamento das funções de onde a chamada do panic! veio para descobrir em que parte do código está o problema
- Tentar acessar um elemento de um vetor que não está dentro do seu tamanho vai levar o Rust vai proteger o programador dessa vulnerabilidade e vai parar a execução e não vai continuar a executar
- Podemos mudar o valor de uma variável de ambiente chamada RUST_BACKTRACE para ter um erro mais verboso e que vai mostrar mais detalhes do erro que ocorreu

9.2 Erros recuperáveis com *Result* (*result*)

- A maioria dos erros que acontecem não são tão sérios, não sendo necessário a paralização do programa
- Algumas vezes, quando uma função falha, é por uma razão que podemos facilmente interpretar e responder
- Podemos definir um enum Result do Rust da seguinte maneira:

```
//os tipos T e E sao tipos genericos
//o T representa o tipo que sera retornado em sucesso
//e o E em caso de erro
enum Result<T, E>{
    Ok(T),
    Err(E),
}
```

- Já que o Result usa parâmetros genéricos, podemos usar o tipo Result e funções definidas nele em várias situações diferentes, em casos onde queremos que o valor retornado em caso de sucesso e erro sejam diferentes

- O match abaixo usa a macro panic! para qualquer tipo de falha. No entanto podemos querer tomar diferentes decisões de acordo com a falha que aconteceu

```
use std::fs::File;

fn main() {
    let greeting_file_result =
    File::open("hello.txt");

    let greeting_file = match
    greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem
        opening the file: {:?}", error),
    };
}
```

- No exemplo que temos dentro do cargo checamos que tipo de erro ocorreu até e fazemos coisas de acordo com isso
- Mais para frente veremos como lidar com esse Result de uma maneira menos verbosa
- Usar o match pode funcionar, mas ser um pouco verboso, então além da opção que daremos mais a frente, o próprio tipo Result<T, E> possui métodos definidos para fazer várias tarefas mais específicas
- O método unwrap é um atalho que funciona da mesma maneira do código escrito aqui acima no trecho de código, ou seja, caso Ok, o retorno será do valor dentro de Ok e caso Err, o unwrap chamará o panic!
- Temos também o expect que nos deixa escolher a mensagem do panic!, podendo ser muito útil para passar uma boa mensagem de erro caso ocorra
- Em códigos de Rust com mais qualidade geralmente se usa mais o expect do que o unwrap para dar mais contexto sobre as falhas das operações
- O rust permite que chamadas de função consigam propagar o erro utilizando o resultado para que aquele que chamou decida o que fazer
- Esse padrão de propagação de erro é tão comum que o Rust tem um operador que o deixa mais fácil com o ?
- Colocar um ? depois de um valor Result em uma expressão equivale a fazer uma expressão match para lidar com os valores do Result
- Existe uma diferença entre usar a expressão match e o operador ?: Os valores de erro que tem o operador ? chamados a ele passam por uma função from definida na biblioteca padrão
- Isso faz com que o tipo de erro recebido na função from seja convertido para o tipo de retorno de erro na função em questão, sendo útil quando a função retorna um tipo de erro para representar todas as maneiras que a função pode falhar
- Ler de um arquivo para uma string é comum o suficiente para termos uma função já definida em fs::read_to_string()
- O operador ? só pode ser usado em funções cujo tipo de retorno é compatível com o valor usado em ?. Isso porque o operador faz uma retorno anterior de um valor da função, assim como os braços das expressões de match
- Podemos usar o ? tanto para Option quanto para Result. Podemos usar em funções que retornam esses tipos em questão mas não podemos misturar um com o outro

9.3 Usar ou não *panic*! eis a questão!

- Agora precisamos decidir quando devemos usar o panic! e quando demos apenas retornar o Result
- Temos que lembrar que quando tomamos a decisão de usar o panic! para um erro em qualquer situação, ele será irrecuperável, ou seja, se estamos em uma função, estamos tomando uma decisão ativa que não podemos continuar com o código
- Quando escolhemos o Result, estamos dando a quem chamou opções, ou seja, ele pode tentar se recuperar do erro ou perceber que é impossível e então chamar o panic!

- Portanto retornar o Result é uma boa escolha padrão ao definir uma função que pode falhar

- Em teste, se um método falha, queremos que o teste inteiro falhe, mesmo que o método em questão não seja o testado, então nesses casos talvez seja melhor usar o panic!

- A sugestão é fazer o código entrar em pânico quando temos há a possibilidade dele acabar em um estado ruim, nesse caso, quando alguma suposição, garantia, contrato ou invariante é quebrada

- Se alguém chama seu código e passa valores que não fazem sentido, é melhor retornar o erro para que o usuário possa decidir o que fazer no caso

- Porém em casos que continuar possa ser inseguro ou problemático, a melhor escolha é chamar o panic! e alertar o usuário do seu código

- De modo similar o panic! é também apropriado se chamamos código externo que esta fora de nosso controle e retorna estados inválidos que não podemos arrumar

- No entanto, quando uma falha é esperada, é mais apropriado usar o Result que fazer a chamada de panic!

- Quando seu código faz operações que pode colocar o usuário em risco se usar valores inválidos, seu código deve verificar e entrar em pânico caso os valores não sejam válidos. Operar com dados inválidos pode expor seu código para vulnerabilidades

- O sistema de tipos do Rust já consegue fazer algumas checagens para nós. No caso de Option por exemplo, pode assegurar que sempre temos algum valor la, ou no caso de um u32, temos certeza de que não haverá negativos

10. Genéricos, características e tempo de vida.

10.1 Tipos de dados genéricos

- Usamos genéricos para criar definições de itens como assinaturas de funções ou structs, onde podemos usar vários tipos de dados concretos diferentes

- Imagine que queremos fazer uma função genérica que conseguiria pegar um vetor de qualquer tipo e retornar o maior elemento do vetor. Imagine que tentássemos algo desse tipo:

```
fn maior<T>(lista : &[T]) -> &T {

    let mut maior = &list[0];

    for item in list {
        if item > maior {
            maior = item
        }
    }

    maior
}
```

- O código acima não iria funcionar uma vez que nem todos os tipos tem esse tipo de comparação >, portanto o compilador soltaria um erro

- Em C++ existem templates que fariam essa função passar pelo compilador mas caso passássemos um parâmetro incorreto o compilador daria erro. O Rust exige que o programador demonstre as capacidades esperadas do tipo genérico a ser definido

- Em Java, os objetos tem métodos padrões que poderiam fazer isso acontecer, porém o Rust não tem tais definições. Sem restrições, um tipo T genérico não tem capacidade: não pode ser impresso, clonado ou mudado

- Podemos definir o uso de tipos genéricos em structs usando a sintaxe <>

```
struct Ponto<T> {
    x: T,
    y: T,
}
```

```
//funciona!
let inteiro = Ponto { x: 5, y: 10};

//nao funciona
let errado = Ponto { x: 5, y: 4.0};
```

- Podemos ver que a sintaxe é similar ao que usamos numa declaração de função. Note que o código acima está usando o mesmo tipo genérico T para ambos x e y, não podendo ser um tipo diferente
- Podemos sim definir com tipos genéricos, dois diferentes para cada variável do ponto por exemplo:

```
struct Ponto<T, U> {
    x: T,
    y: U,
}
//agora o exemplo que nao funcionaria lá
//de cima funcionaria sem problemas
```

- Podemos usar quantos parâmetros genéricos quisermos em nossas implementações só que isso pode tornar o código ilegível, então temos que tomar cuidado
- Podemos usar não só com structs mas também para enums. Talvez a definição de um Option<T> e Reuslt<T, E> faça mais sentido agora

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- Caso perceba que existem situações em seu código com múltiplos structs e enums que tem mesma definição com tipos diferentes, podemos evitar a duplicação usando tipos genéricos
- Podemos usar genéricos da implementação de métodos de enums e structs:

```
struct Ponto<T> {
    x: T,
    y: T,
}

//declaramos T depois de impl
//para especificar que estamos implementando
//no tipo Ponto<T>
impl<T> Ponto<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

//esse metodo so esta disponiveis para
//instancias do tipo Ponto<f32> e nao Ponto<T>
impl Ponto<f32> {
    fn distancia_origem(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

//tome cuidado porque agora nao podemos definir um
//dintancia_origem para o Ponto<T>
//isso iria gerar um erro pois o Rust nao saberia
//qual metodo chamar o do tipo T generico ou o f32
```

- Os tipos genéricos não fazem o programa rodar mais devagar comparado a tipos concretos. O Rust consegue isso tornando todo código genérico em específico preenchendo tipos concretos em tempo de compilação
- Nossos tipos genéricos são analisados e o compilador cria versões dos tipos específicas que podem ser usadas

10.2 Traços: definindo comportamentos comuns (*traits*)

- Traços são funcionalidades que um tipo particular pode compartilhar com outros
- Podemos usar esses traços para definir um comportamento em comum. Podemos usar limites de traços para especificar que um tipo genérico pode ser qualquer um que tenha um certo comportamento

- Traços aqui são o que comumente chamamos de interfaces em outras línguas só que com algumas diferenças
- Um comportamento de um tipo são os métodos que podemos chamar no tipo
- Tipos compartilham o mesmo comportamento se podemos chamar os mesmos métodos em ambos
- Podemos implementar traços locais em tipos locais e em tipos externos e só podemos implementar traços externos em tipos locais. Não podemos implementar traços externos em tipos externos pois não estamos no local onde são definidas
- Dê uma olhada no exemplo para entender melhor
- Traços e limites de traços nos deixam reduzir código duplicado e especificar ao compilador o que queremos em nossas funções

10.3 Validando referências com tempo de vida

- O tempo de vida e é outro tipo de genérico que já usávamos. Em vez de garantir que um tipo tivesse o comportamento desejado, o tempo de vida garante que a referência é válida o tempo necessário
- Toda referência em Rust tem um tempo de vida, que é o escopo no qual aquela referência é válida. Na maioria das vezes esse tempo de vida é inferido ou implícito
- Precisamos denotar o tempo de vida de referências quando elas podem estar relacionadas de certas maneiras diferentes
- Anotar o tempo de vida de uma variável é um conceito bem diferente de outras linguagens, então essa seção vai servir para familiarizar um pouco de como ele pode aparecer
- O principal propósito do tempo de vida é prevenir referencias que ficam penduradas, que fazem o programa referenciar dados que não são o que queremos. Considere o trecho de código abaixo por exemplo

```
fn main() {
  // variavel r sem valor inicial
    let r;
    {
        let x = 5;
        //atribuindo a r uma referencia de x
    r = &x;
    }
    //gerariamos um erro porque a referencia de x
    //nao vive tempo o suficiente por estar dentro
    //do escopo acima!
  println!("r: {}", r);
}
```

- Acima, a variável r "vive por mais tempo" do que x, dado o escopo aninhado em que x esta. Ou seja, r é válido no escopo maior enquanto que o valor de x já foi liberado. Se r fosse se tornar uma referencia de x, que em seu escopo já foi liberada, qualquer coisa que tentássemos fazer com r não funcionaria direito
- A notação de tempo de vida não muda a longevidade de uma referencia. Na verdade, descrevem as relações do tempo de vida de diversas referencias uma com as outras sem afetar os tempos de vida
- Assim como funções podem aceitar qualquer tipo quando uma assinatura especifica um parâmetro genérico, funções aceitam referencias com qualquer tempo de vida se dermos um parâmetro com tempo de vida genérico
- A notação de tempo de vida é bem não usual: o nome de parâmetros de tempo de vida precisam começar com apostrofe(') e usualmente pequenos com todas as letras minúsculas
- Alguns exemplos da notação:

```
&i32 // Uma referencia
&'a i32 // uma referencia com um
        //tempo de vida explicito
&'a mut i32 //uma referencia mutavel de um
            // tempo de vida explicito
```

- A anotação sozinha não possui muito significado porque elas estão la para dizer ao Rust como os parâmetros de tempo de vida de múltiplas referencias se relacionam
- Vamos agora dar um exemplo de como ele funcionaria em uma função:

```
// explicando a assinatura de funcao
// definimos um tempo de vida 'a
// e definimos o seguinte
// o tempo de vida de x e o mesmo do de y
// assim como o retorno da funcao tem que
//ter o mesmo tempo de vida que ambas as
//variaveis. Ou seja, x, y e nosso retorno
//precisam possuir o mesmo tempo de vida
fn maior_string<'a>(x: &'a str, y: &'a str) ·
    if x.len() > y.len(){
        x
    } else {
        y
    }
}

//poderiamos usar da seguinte maneira
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = maior_string
    (string1.as_str(), string2);
    println!("A maior string é {}",result);
}
```

- Note que a função pega fatias de strings, ou seja, não vai tomar posse dos parâmetros
- Quando fazemos esse tipo de especificação, não estamos mudando o tempo de vida dos valores, estamos apenas especificando para o verificador de empréstimo que deve rejeitar valores que não aderem essas condições
- A função não precisa sabe exatamente quanto tempo x e y vão viver, somente que algum escopo pode ser substituído por 'a que satisfaz a assinatura
- O notação do tempo de vida vira parte do contrato de uma função, assim como assinatura de tipos
- A sintaxe de tempo de vida tem como objetivo conectar o tempo de vida de parâmetros e retornos de funções.
- Uma vez conectados, o Rust tem informação o suficiente para permitir operações segura em memória e proibir operações que levariam a ponteiros soltos ou violariam a segurança da memória
- Podemos definir structs que possuem referencias, mas para isso temos que adicionar a notação de tempo de vida em cada referencia na definição do struct

```
struct ExcertoLegal<'a> {
    parte: &'a str,
}

impl<'a> ExcertoLegal<'a> {
    fn nivel(&self) -> i32 {
        3
    }
}

fn main() {
    let novela = String::from("Shadow Wizard
    Money gang. We love casting spells!");
    let primeira_frase = novel.split('.').
    next().expect("Não achei um .");
    let i = ExcertoLegal {
        parte: primeira_frase
    }
}
```

- Um tempo de vida especial que precisamos discutir é o 'static, que denota que uma referencia pode viver por toda a duração do programa
- Todas as strings literais possuem esse tempo de vida que podemos anotar como o seguinte:

```
let s: &'static str = "Meu tempo de vida é estatico";
```

- Vamos agora juntar tudo que aprendemos em uma única assinatura de função: tipos genéricos, limites de traços e tempos de vida

```
use std::fmt::Display;

fn maior_com_anuncio<'a, T> (
    x: &'a str,
    y: &'a strm
    anun: T,
) -> &'a str //retorna uma string com
            //tempo de vida 'a
where
    //O tipo T precisa implementar Display
    T: Display,
{
    //usando o Display especificado no where
    println!("Anuncio!!! {}", ann);
    //funcao de maior que
    //ja definimos na secao
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

10.4 Inventário de posse #3

- Quiz para reforçar conhecimento sobre posse encontrado aqui

11. Sobre testes, para garantir que sua lógica de programação esteja correta

11.1 Como escrever testes (*lets-test*)

- Funções de teste verificam o funcionamento de trecho de código para ver se ele está de comportando da maneira correta

- Geralmente 3 ações são feitas: configuração de dados ou estados se necessário, acionar o código que queremos testar e comparar os resultados com o esperado

- O Rust nos da alguns recursos para escrever testes que incluem o atributo test, alguns macros e o atributo should_panic

- Em sua maneira mais simples, uma teste em Rust é uma função com uma anotação de atributo test

- Atributos são meta-dados sobre pedaços de códigos em Rust; um exemplo é o atributo derive que já até utilizamos

- Para tornar uma função já existente em função de teste, apenas adicionamos #[test] na linha anterior a função fn

- Quando criamos um novo projeto como biblioteca com o Cargo, um módulo de teste automaticamente é criado

- O comando cargo test roda todos os teste de um projeto

11.2 Controlando como os testes são rodados

- Assim como o cargo run compila o código e roda o binário resultante, o cargo test compila seu código e roda o binário resultante

- O comportamento padrão do cargo test é rodar todos os testes em paralelo e capturar os resultados durante os testes, a omitindo resultados específicos dos testes para mais fácil leitura

- Podemos especificar opções de linha de comando para mudar esse comportamento padrão

- Algumas opções são dadas ao comando cargo test, enquanto outras são para o teste binário resultante

- Podemos listar os argumentos para o comando seguido do separador – para os argumentos do binário

- Para as opções do comando podemos usar cargo test –help e para as opções do binário podemos usar cargo test – –help

- Se não quisermos que os testes rodem em paralelo podemos usar o argumento –test-threads e passar apenas uma thread como argumento

```
cargo test -- --test-threads=1
```

- Por padrão o Rust vai capturar tudo que é produzido na saída padrão, como por exemplo, um println! se quisermos ver essas saídas podemos usar o argumento –show-output

```
cargo test -- --show-output
```

- Podemos executar apenas um único teste passando o nome do teste específico na linha de comando, assim como podemos passar uma parte do nome dos testes que queremos usar

```
cargo test <nome_do_teste>

cargo test <substring_dos_nomes>
```

- Para testes muito custosos que não precisam rodar toda vez que testamos nosso código podemos usar o atributo ignore nesses testes

```
#[test]
#[ignore]
fn teste_custoso {
    //demora...
}

//para rodar os ignorador podemos fazer
cargo test -- --ignored

//ou se quisermos rodar todos podemos fazer
cargo test -- --include-ignored
```

11.3 Organização dos testes (lets-test)

- A comunidade do Rust pensa nos testes em duas categorias: testes unitários e de integração
- Os unitários são menores e focados, testando um módulo isolado por vez e pode testar interfaces privadas
- Testes de integração são completamente externos da biblioteca e usam o código assim como qualquer outro código externo, usando as interfaces públicas e testando potencialmente muitos módulos por vez
- Testes unitários testam cada unidade do código isoladamente do resto do código para ser mais fácil de identificar onde o código pode não estar funcionando
- Podemos colocar esses testes dentro do diretório src em cada arquivo com código sendo testado. O mais comum é criar um módulo chamado tests e usar a notação cfg(test) no módulo
- O #[cfg(test)] diz ao Rust para compilar e rodar os testes apenas quando rodamos o cargo test e não o cargo build por exemplo
- O Rust possui regras de privacidade que permitem que testemos funções privadas ou seja podemos fazer algo como abaixo

```
//funcao privada sem a keyword pub
fn soma_interna(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    //trazemos todos os modulos do pai para
    //dentro do modulo tests o que nos deixa
    //usar as funcoes privadas até
    use super::*;

    #[test]
    fn interno() {
      assert_eq!(4, soma_interna(2, 2));
  }
}
```

- Criamos um diretório tests no nível mais acima do nosso projeto. O Cargo sabe que deve olhar dentro desta pasta para arquivos de teste
- Podemos criar vários arquivos e o Cargo compila cada um deles como uma crate individual
- Não podemos fazer testes de integração em crates binárias, apenas de biblioteca, porque somente funções de bibliotecas são expostas para outras crates usarem; crates binárias devem ser executadas sozinhas

12. Construímos nossa implementação de uma subconjunto da funcionalidade de um comando *grep*. (*minigrep*)

12.1 Construindo o projeto

Nesse capítulo, foi implementado um projeto guiado pelo livro de Rust, com os seguintes passos e funcionalidades:

- Aceitando argumentos de linha de comando
- Lendo um arquivo
- Refatorando para melhorar modularização e manipulação de erros
- Usando TDD para desenvolver a funcionalidade da biblioteca
- Trabalhando com variáveis de ambiente
- Escrevendo mensagens de erro para Standard Error em vez de Standard Output

Tudo foi feito no projeto e organizado por commits. O projeto feito pode ser acessado pelo link: https://github.com/VitorKenzo/rust-minigrep

13. Explora fechamentos e iteradores, características que vem de linguagens funcionais.

13.1 Fechamentos (*closure*): Funções anônimas que capturam o ambiente (*closures*)

- Fechamentos do Rust são funções anônimas que podemos salvar em uma variável ou passar como argumento para outras funções
- Diferente de funções, Fechamentos podem capturar valores do escopo onde foram definidos
- Fechamentos geralmente não precisam de notação para os tipos dos parâmetros ou o valor de retorno como em funções definidas com fn
- Fechamentos não são usados em interfaces expostas como de funções, são armazenadas em variáveis e usadas sem serem nomeadas, não sendo expostas para os usuários da biblioteca
- Fechamentos são geralmente curtos e relevantes dentro de um contexto pequeno. Dentro desses contextos, o compilador pode inferir os tipos dos parâmetros e o tipo de retorno
- Podemos adicionar uma notação para o tipo das variáveis se quisermos aumentar a claridade do trecho ao custo de ser mais verboso do que é necessário

```
let fechamento_caro = |num : i32| -> i32 {
  println!("Estou pegando no sono...");
  thread::sleep(Duration::from_secs(2));
    num
};
```

- Com isso os fechamentos começam a ficar mais parecidos com funções. Perceba o quão similares podem ficar com o trecho abaixo:

```
fn soma_um_v1 (x: u32) -> u32 { x + 1 }
let soma_um_v2 = |x: u32| -> u32 { x + 1 };
//v3 e v4 vao ter tipos inferidos pelo uso
//por so ter uma instrucao n precisamos dos {}
let soma_um_v3 = |x| { x + 1 };
let soma_um_v4 = |x| x + 1;
```

- Ao definir fechamentos o compilador infere um tipo concreto para cada parâmetro e para o valor de retorno
- No exemplo abaixo, quando chamamos o fechamento pela primeira vez com um tipo String o compilador infere o tipo do fechamento como tal e causaríamos um erro ao chamar com um inteiro por exemplo

```
//apenas retorna o valor
let fechamento_bobo = |x| x;

let s = fechamento_bobo(String::from("oi"));
//let n = fechaento_bobo(5); <- causaria um erro
```

- Fechamentos capturam valores de 3 maneiras: por empréstimo imutável, empréstimo mutável e tomando pra si
- O fechamento decide como será feito baseado em seu corpo
- Uma vez que o fechamento captura uma referência ou toma posse de um valor do ambiente onde o fechamento é definido, o código do fechamento vai definir o que acontece
- O corpo de um fechamento pode mover um valor capturado para fora do fechamento, mudar o valor capturado, não fazer nada disso ou não capturar nada do ambiente pra começo de conversa
- O fechamento pode implementar 3 características, FnOnce, FnMut e Fn dependendo do seu corpo
- FnOnce são para as que só podem ser chamadas uma vez, uma que move os valores capturados só implementa essa característica
- FnMut se aplica aqueles que não movem os valores capturados mas podem alterar os valores
- Fn se aplica a fechamentos que não mudam ou movem os valores capturados

13.2 Processando uma série de itens com um iterador

- Iteradores permitem que façamos uma tarefa em uma sequência de itens de cada vez, sendo responsáveis pela lógica de iteração por cada item e determinar quando a sequencia termina
- Iteradores em Rust são preguiçosos, ou seja, eles não possuem efeito até chamarmos métodos que consumam o iterador
- Todos os iteradores implementam uma característica chamada Iterator que é definida na biblioteca padrão
- A característica apenas precisa da definição de um método next, onde retorna um item do iterador dentro de um Some e quando o iterador termina retorna um None
- Métodos que chamam essa função next usam os iteradores, por exemplo o método sum que toma posse de um iterador e usa ele para iterar pelos itens chamando next
- A função map toma como parâmetro um fechamento e retorna um novo iterador que produz as modificações nos itens

```
let v1 = vec![1, 2, 3];

let v2 = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

- O método collect consome o iterador e coleta os valores resultantes em um tipo de dado de coleção
- Como o map toma um fechamento, podemos dar qualquer operação para ser feita por item, sendo bem customizável

13.3 Melhorando a entrada e saída (I/O) do nosso projeto

- Nessa seção, foi feito um **novo commit no minigrep** com base nas coisas aprendidas no capítulo 13.

13.4 Comparando a performance: *Loops vs* Iteradores

- Fazendo um benchmark de ambas implementações provavelmente vamos chegar em um resultado onde o iterador se mostra um pouco melhor
- O ponto que queremos fazer aqui é que iteradores, por mais que sejam uma abstração de alto nível, são compiladas para um código similar a uma iteração feita em baixo nível pelo programador
- Iteradores são uma das abstrações de zero custo do Rust
- Portanto, podemos usar fechamentos e iteradores sem medo de penar em desempenho

14. Examinaremos o *Cargo* a fundo e melhores práticas para compartilhar bibliotecas com outros.

14.1 Customizando *builds* com perfis

- Em Rust temos perfis customizáveis com diferentes configurações que permitem que o programador tenha mais controle sobre as opções de compilação de código chamado de release profiles
- O Cargo tem 2 perfis principais, o dev quando rodamos o cargo build e o release quando usamos cargo build –release
- O Cargo possui configurações padrão para cada perfil que não foi explicitamente adicionado no arquivo Cargo.toml
- Podemos customizar adicionando seções como [profile.*] e sobrescrever opções que quisermos

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

- Para a lista completa das opções de configurações veja a documentação aqui

14.2 Publicando um caixote no *crates.io*

- Em alguns exemplos que fizemos, trouxemos dependências de pacotes do crates.io para o nosso projeto, mas podemos também compartilhar nosso código publicando
- O Rust e o Cargo tem ferramentas que tornam mais fácil a publicação de pacotes para pessoas encontrarem e usarem
- Documentar precisamente seus pacotes ajuda outras usuários entenderem a como usar ele, então vamos passar um algumas dicas de como documentar
- O Rust tem um tipo particular de comentário específico para documentação, que gera uma documentação em HTML
- Comentários de documentação usam /// em vez de dois e tem suporte para a notação em Markdown para formatação do texto
- Coloque a documentação antes do item que esta documentando como no exemplo abaixo

````
/// Soma um ao número dado
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::soma_um(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn soma_um(x: i32) -> i32 {
    x + 1
}
````

- Podemos gerar a documentação através da linha cargo doc. Esse comando roda a ferramenta rustdoc que é distribuída junto com o Rust e gera o HTML em target/doc
- Por conveniência podemos rodar cargo doc –open ele cria um HTML para o caixote em questão e toda a documentação para as dependências também e abre no navegador
- Autores de caixotes geralmente usam algumas seções em comum para manter um padrão:
  - Pânicos: A seção de Panics são os cenários onde a função documentada pode gerar o pânico
  - Erros: Se a função retorna um Result, que tipos de erros podem ocorrer e quais condições podem causar os erros
  - Segurança: Se a função não for segura, deve ter uma seção explicando o porque e cobrindo as invariâncias que a função espera de quem a chama
- Colocar exemplos de blocos de código pode ajudar a demonstrar como usar a biblioteca e além disso tem um bonus, o cargo test pode rodar esses trechos como testes de documentação

- O estilo de comentário //! adiciona documentação para o item que contém os comentários. Utilizado geralmente dentro do arquivo raiz, src/lib.rs por exemplo, para documentar o caixote como um todo
- São úteis para descrever caixotes e módulos, explicando o propósito geral do container
- Uma estrutura para sua API pública tem que ser levada em conta ao publicar um caixote. Já vimos o funcionamento da palavra chave pub e como trazer itens para o escopo usando o use
- Se a sua estrutura interna de struct estiver complexa e for difícil de utilizar, não é necessário modar essa estrutura, podemos apenas re exportar itens para fazer uma estrutura pública diferente da privada usando pub use
- A documentação vai mostrar essas re exportações sem alterar a estrutura interna de como estão sendo utilizadas
- O pub use te da flexibilidade em como estruturamos os caixotes internamente e desacoplamos o uso interno do que é apresentado para o usuário
- Antes de publicar caixotes, precisamos criar uma conta no crates.io e conseguir um API token, para fazer isso basta visitar o a página inicial do crates.io e logar através de uma conta no GitHub
- Com isso pode-se recuperar uma chave de API em http://crates.io.me que podemos colocar como argumento do comando cargo login
- O comando informa o Cargo do seu token e armazena ele localmente
- Para publicar um caixote, primeiro é válido colocar alguns meta dados no Cargo.toml, algo parecido com o que esta abaixo

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you gues
what number the computer has chosen."
license = "MIT OR Apache-2.0"
```

- A [documentação] do Cargo descreve outros metadados que podemos colocar
- Tome cuidado porque uma publicação é permanente, a versão nunca poderá ser sobrescrita e o código não poderá ser deletado
- O crates.io tem como um de seus objetivos ser um grande arquivo permanente de código para que todos os projetos que dependem de caixotes do site continuem a funcionar
- Para publicar um código basta rodar cargo publish e esta feito
- Para publicar uma nova versão basta trocar o campo version do Cargo.toml e republicar. Use como guia para as versões [este] site
- Por mais que não possa remover versões antigas de caixotes, podemos prevenir que projetos futuros adicionem ele como nova dependência
- Podemos puxar uma versão para fazer isso utilizando o comando cargo yank e especificando a versão que queremos retirar: cargo yank –vers 1.0.1
- Podemos desfazer isso colocando um argumento –undo no final do comando

14.3 Espaços de trabalho para o *cargo* (*workspace-add*)

- Conforme um projeto cresce, pode ser que voce queira dividir sua biblioteca em vários diferentes caixotes
- O Cargo oferece as áreas de trabalho (workspaces) que podem ajudar na manutenção de diversos pacotes relacionados
- Uma área de trabalho é um conjunto de pacotes que compartilham o mesmo *Cargo.lock* e diretório de saída
- No exemplo iremos mostrar um jeito comum de trabalhar com um workspace

- Para rodar um caixote específico dentro do workspace precisamos usar o argumento -p e passar o nome como argumento: cargo run -p adder

14.4 Instalando binários do *crates.io* com *install*

- O comando cargo install permite que instalemos para uso caixotes binários localmente
- É apenas uma maneira conveniente para que desenvolvedores instalem ferramentas que outros compartilharam no crates.io
- Todos os binários instalado através do cargo install são armazenados na raiz da instalação na pasta *bin*

14.5 Estendendo *cargo* com comandos customizados

- O Design do cargo é feito para que possamos estender ele com novos sub comandos sem ter que modificar o Cargo
- Comandos customizados aparecem quando fazemos um cargo –list
- Poder instalar extensões com o cargo install e usar essas ferramentas como se fossem do próprio Cargo é um dos benefícios desse design

15. Ponteiros inteligentes disponíveis na biblioteca padrão.

15.1 Introdução

- Um ponteiro é um conceito geral para uma variável que contém o endereço de memória, e esse endereço aponta para algum outro dado
- O ponteiro mais comum de se encontrar em Rust é um referencia que já vimos, sendo indicadas por & e pegam emprestado o valor para o qual apontam
- Ponteiros inteligentes por outro lado, são estruturas que agem como ponteiros mas possuem meta dados adicionais
- O Rust possui uma variedade de ponteiros inteligentes definidos na biblioteca padrão que dão funcionalidades além de referencias comuns
- Com os conceitos de posse e empréstimo, o Rust tem uma diferença entre referencias e ponteiros inteligentes, referencias podem pegar emprestado dados em muitos casos, mas ponteiros inteligentes tomam posse dos dados que apontam
- Os ponteiros inteligentes são implementados geralmente por structs, que implementam funcionalidades de Deref e Drop
- O Deref permite que uma instância de ponteiro inteligente se comporte como uma referencia
- O Drop permite customizar o que acontece quando o ponteiro inteligente sai do escopo
- Nessa seção vamos cobrir apenas os ponteiros inteligentes mais comuns da biblioteca padrão: Box<T>, Rc<T>, Ref<T> e RefMut<T> que são acessados por RefCell<T>, um tipo que reforça as regras de empréstimo em tempo de execução ao invés de compilação
- Vamos discutir também padrões de mutabilidade interna, onde um tipo imutável expões uma API para mudar seu valor interno e ciclos de referencia e como podem gerar vazamento de memória

15.2 Usando *Box<T>* para apontar para dados no *heap*

- O ponteiro inteligente mais direto que temos em Rust é uma caixa (box) cujo tipo escrevemos como Box<T>
- Essa caixas permitem o armazenamento de dados no heap ao invés da pilha. O que fica na pilha é o ponteiro para os dados em heap
- Usamos esses ponteiros geralmente em 3 situações distintas:

- Quando temos um tipo que não saberemos o tamanho em tempo de compilação e queremos usar o valor num contexto que requer essa exatidão
  - Quando temos uma grande quantidade de dados e queremos dar posse mas assegurar que os dados não serão copiados ao fazer isso
  - Quando queremos tomar posse de um valor e nos importamos apenas que ele implemente uma certa funcionalidade em vez de ser um tipo específico

- Vamos dar um exemplo só para entendermos um pouco melhor a sintaxe

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

- Um valor de tipo recursivo pode ter outro valor do mesmo tipo como parte de si mesmo. Isso é desafiador porque o compilador não tem como saber quanto espaço ele vai ocupar em compilação
- Digamos que temos um tipo List que pode ter dentro de si em um de seus argumentos uma outra variável do tipo List
- O compilador nunca deixaria um código desse rodar sem ponteiros, porque ele não teria como definir o tamanho da variável em tempo de compilação
- Para conseguir isso podemos usar os ponteiros para que ele apenas precisa do ponteiro em alocação e o resto da estrutura fique em heap

15.3 Tratando ponteiros inteligentes como ponteiros normais com a característica *Deref*

- A característica de Deref nos permite customizar o comportamento de ponteiros inteligentes na hora de sua deferenciação com o operador *
- Essa característica permite que ponteiros inteligentes sejam tratados como referencias normais do Rust
- Uma referencia normal é um tipo de ponteiro, e podemos pensar nele como uma seta para o valor armazenada em algum lugar
- Podemos criar uma referencia para um valor i32 e então usar o operador de deferência para seguir a a referência até seu valor

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

- Poderíamos fazer algo bem similar utilizando o tipo Box<T> em vez de uma referencia, o operador de deferenciação funciona da mesma maneira do código acima:

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

- A grande diferença é que neste y temos uma instância de um Box<T> apontando para uma cópia do valor de x ao invés de termos uma referencia apontando para o valor de x.
- Abaixo temos uma implementação simples da característica de Deref:

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
```

```
        &self.0
    }
}
```

- Em Rust podemos ter uma conversão implícita de deferências para todos os tipos que implementam a característica de Deref
- Esse tipo de conversão foi adicionado para que fosse mais fácil escrever chamadas de função sem usar muitas referencias e deferências com & e *
- Quando a característica de Deref é definida nos tipos envolvidos, o Rust analisa os tipos e usa Deref::deref o quanto precisar para que as referencias batam com o tipo do parâmetro em questão
- Assim como usamos o Deref para o operador * em referencias imutáveis, usamos o DerefMut para o mesmo operador em referencias mutáveis

15.4 Rodando código em limpeza com a característica *Drop*

- A segunda característica importante de ponteiros inteligentes é o Drop, que nos deixa customizar o que acontece quando o valor vai sair do escopo
- Introduzimos o Drop nesse contexto porque ele é quase sempre utilizado com ponteiros inteligentes, mas pode ser usado quando estamos soltando recursos como arquivos ou conexões
- No Rust, podemos especificar o trecho de código que deve ser rodado sempre que um valor sai do escopo, e o compilador vai inserir esse código automaticamente
- Como resultado, não precisamos tomar cuidado com código de limpeza pelo código e ainda não iremos vazar nenhum tipo de recurso
- Isso pode ser feito quando implementamos a característica de Drop, requerendo uma implementação do método drop que tem como parâmetro uma referencia mutável do objeto

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer
        with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

- O Rust automaticamente chama o drop para nós quando as instâncias saem do escopo, chamando o código que especificamos
- Vamos dizer então que você queria fazer essa limpeza antes do que o compilador julgue como necessário. Podemos fazer isso usando método drop manualmente, chamando de std::mem::drop
- A função acima é diferente do método drop dentro do Drop. Chamamos ela passando um argumento que é o valor que queremos liberar. A chamada seria algo como:

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c); //aqui
    println!("CustomSmartPointer dropped
            before the end of main.");
}
```

- Não precisamos nos preocupar com o problema de acidentalmente limpar valores em uso: o sistema garante que as referencias

sempre são válidas e que o drop é chamado apenas uma vez

15.5 *Rc<T>*, O ponteiro inteligente com referencia contada

- Na maioria dos casos, a posse em Rust é clara, sabemos exatamente que variável tem posse de que valor
- Porém existem estruturas que um valor pode ter vários proprietários
- Imagine num grafo por exemplo, várias arestas podem apontar para um mesmo nó e portanto, todos esses teriam posse dele. Além disso, o nó não pode ser limpo enquanto houverem referências a ele
- Em Rust temos que explicitamente habilitar posse múltipla com o tipo Rc<T>, uma abreviação de reference counting ou contagem de referencia
- O tipo acima mantém o número de referencias que um valor possui para determinar se o valor esta em uso ainda ou não
- Usamos o tipo quando queremos alocar dados no heap para várias partes de nosso programa ler e não sabemos qual parte termina por último de usar os dados
- Só podemos usar o Rc<T> em cenários com uma única thread
- Precisamos usar um use para trazer o Rc<T> para o nosso escopo com std::rc::Rc
- Com ele no escopo podemos usar a função Rc::clone(&x), que faz uma não faz uma cópia profunda de x, mas sim aumenta a contagem de referencias a ele
- Por referencias imutáveis, o Rc<T> permite o compartilhamento de dados por várias partes do programa em modo somente de leitura
- Se fosse possível com referencias mutáveis também, poderíamos acabar violando as regras de empréstimo estabelecidas

15.6 *RefCell<T>* e o padrão interno de mutabilidade

- Mutabilidade interior é um padrão de design em Rust que permite a mutação de dados mesmo com referencias imutáveis para o dado
- Para fazer isso, o padrão usa código não seguro, unsafe, dentro de um estrutura de dados para quebrar as regras normais do Rust
- Só podemos usar esses tipos de padrões quando podemos assegurar que as regras de empréstimo serão seguidas em execução
- O tipo RefCell<T> representa posse única sobre os dados que possui, só que a diferença desse tipo é que as regras de empréstimo são reforçadas em tempo de execução
- Ou seja, se as regras forem quebradas o programa entra em pânico e sai
- A vantagem de checar as regras em tempo de execução é que certos cenários seguros de memória podem acontecer, que seriam negados pelas checagens em tempo de compilação
- A análise estática do compilador é conservadora, sendo que algumas propriedades do código são impossíveis de detectar por análise de código
- O tipo RefCell<T> é útil quando temos certeza que um código segue as regras de empréstimo mas o compilador não consegue entender e garantir isso
- Só podemos usar esse tipo em cenários com uma única thread
- Vamos dar um recap de quando escolher os tipos Box<T>, Rc<T> e RefCell<T>:
  - O Rc<T> permite vários proprietários de um mesmo dado, os outros dois só permitem um único
  - Box<T> permite empréstimos mutáveis e imutáveis em tempo de compilação. Rc<T> somente imutável em compilação e RefCell<T> imutável e mutável em tempo de execução
  - RefCell<T> permite a mutação de um valor dentro do tipo mesmo que seja imutável

- O padrão de mutabilidade interior é um onde o código de fora não consegue alterar o objeto mas ele ainda pode de auto mudar
- O RefCell<T> cria referencias novas com métodos borrow e borrow_mut que retornam ponteiros do tipo Ref<T> e RefMut<T> respectivamente
- O RefCell<T> mantém quantos ponteiros do tipo Ref<T> e RefMut<T> estão ativos
- Assim como regras de compilação de empréstimo o tipo em questão só permite ter várias referencias imutáveis e uma única mutável em qualquer ponto da execução
- Se não seguirmos isso, uma mensagem de pânico é gerada pelo tipo
- Uma maneira comum de usar o RefCell<T> é combinando-o com o Rc<T>. Se temos um Rc<T> que segura um RefCell<T>, **podemos ter um valor que pode ter vários usuários e pode ainda ser mutável! ← (Magia negra, tome cuidado!)**

15.7 Ciclos de referencia podem vazar memória

- As regras de segurança do Rust tornam difícil criar memória que nunca é limpa mas não impossível
- Podemos ver que o rust permite vazamentos de memória usando o Rc<T> e RefCell<T>, sendo possível criar referencias onde objetos se referenciam em um ciclo
- Isso gera um vazamento de memória porque a conta de referencias de cada item não chegará a zero por causa do ciclo
- Criar esses ciclos de referencia é difícil, porém não impossível, se temos valores RefCell<T> que contém valores Rc<T> ou combinações similares de tipos que permitem a mutabilidade de interior e contagem de referencias
- Até o momento vimos que o Rc::clone aumenta a contagem forte, strong_count de uma instância de Rc<T> e essa só pode ser limpa quando a contagem chega a zero
- Podemos criar referencias fracas, weak reference, para um valor dentro de uma isntância Rc<T> com Rc::downgrade
- Essas referencias não expressam posse e sua conta não afeta na limpeza da instância, elas seram quebradas assim que a contagem das referencias fortes chegue a zero
- Temos métodos similares para ver a contagem das referencias fracas, como weak_count e temos o tipo Weak<T> de ponteiro inteligente
- Caso essas seja de seu interesse criar estruturas que sejam mais ou menos desses tipo, como listas encadeadas, árvores ou grafos, a sugestão é checar o **[Rustonomicon], porém tenha cautela**

16. Modelos de programação concorrente e como o *Rust* pode ajudar.

16.1 Usando *threads* para rodar código simultaneamente

- Na maioria dos sistemas operacionais atuais, um código de um programa é executado em um processo e o SO faz a manutenção de vários processas de uma vez
- Dentro de programas podemos ter partes independentes que rodam simultaneamente, sendo essas partes o que chamamos de threads
- Dividir a comutação em um programa em várias threads para realizar diversas tarefas pode melhorar performance, mas adiciona complexidade!
- Threads podem gerar problemas exatamente por não sabermos em qual ordem essas threads vão acabar, problemas de corrida, deadlocks ou bugs por exemplo
- Para criar uma nova thread usamos thread::spawn, um função e passamos um fechamento contendo o código que queremos rodar nessa nova thread

- Quando a thread principal de um programa em Rust completa sua execução, todas as threads criadas são desligadas, tendo terminado ou não sua execução

- Como não sabemos como o SO vai escalonar essas threads é possível que uma thread criada nem execute! Para resolver esse problema e o de término de execução prematura podemos usar um JoinHandle

- O JoinHandle é o valor de retorno do thread::spawn e quando chamamos o método join nele, o programa esperava a thread finalizar

```
let handle = thread::spawn(|| {
  for i in 1..10 {
    println!("hi number {} from the spawned
    thread!", i);
      thread::sleep(Duration::
      from_millis(1));
    }
});

handle.join().unwrap();
```

- A chamada de join bloqueia a thread executando atualmente até que a thread representada pelo JoinHandle terminar

- Pequenos detalhes como por exemplo aonde esta a chamada do join podem afetar como as thread vão executar então tome cuidado

- Podemos forçar o fechamento que estamos criando na thread a tomar posse dos valores que vai usar ao invés de deixar o Rust inferir o uso através da palavra move

```
let v = vec![1, 2, 3];

let handle = thread::spawn(move || {
    println!("Here's a vector: {:?}", v);
});
```

16.2 Usando mensagens para transferir dados entre *threads* (*thread-talks*)

- A biblioteca padrão do Rust tem uma implementação de canais que podem ser utilizados para fazer uma comunicação entre duas threads

- Criamos um novo canal usando mpsc::channel, mpsc é um acrônimo para multiple producer, single consumer

- Ou seja, esse canal pode ter vários fins de transmissão porém um único de recepção que consome os valores

- A função que citamos retorna uma tupla, o primeiro elemento é o fim de transmissão (tx) e o segundo o receptor (rx)

- As regras de posse são vitais na hora de enviar mensagens porque nos ajudam a escrever códigos seguros e concorrentes

16.3 Concorrência de compartilhamento de estado

- A troca de mensagens é uma boa maneira de lidar com concorrência, mas não é a única

- Outra maneira seria de múltiplas threads acessarem um mesmo dado compartilhado

- Concorrência com memória compartilhada é como se fosse um posse múltipla, ou seja pode acrescentar complexidade porque diferentes donos fazem a manutenção

- O Mutex - acrônimo para mutual exclusion ou exclusão mutua - permite que apenas uma thread acesse algum dado em um tempo determinado

- O mutex possui um sistema de bloqueio que rastreia quem atualmente tem o acesso exclusivo de um dado e faz isso através de locks

- Precisamos lembrar de 2 regras para usar o mutex:

  - Precisamos primeiro ter o lock em mãos antes de usar os dados

  - Precisamos desbloquear os dados depois de usar para que outros possam usar o lock

- Podemos criar um tipo Mutex<T> usando a função new associada e para acessar o dado, usamos o método lock para adquirir ele

```rust
let m = Mutex::new(50);

{
    let mut num = m.lock().unwrap();
    *num = 9;
}
```

- A chamada de lock falharia se outra thread com o lock entrasse em pânico, o que no nosso caso a nossa entraria junto em pânico por estarmos usando o unwrap
- O Mutex<T> é um ponteiro inteligente que implementa as características de Deref e Drop, ou seja não haverá um esquecimento de esquecer de liberar o lock e bloquear outras threads de usa-lo
- Para termos uma maneira segura de trabalhar com múltiplas posses em situações concorrentes temos o Arc<T> - Acrônimo para atomically reference counted ou referência contada atomicamente
- Esse tipo é seguro para ser compartilhado entre threads, o que não é o caso do Rc<T> por exemplo
- Assim como usamos o RefCell<T> para mudar conteúdos dentro de tipos Rc<T>, usamos o Mutex<T> para fazer o mesmo com o tipo Arc<T>
- O Mutex<T> vem com o perigo de criar deadlocks ocorrendo quando duas threads precisam dos mesmos dois recursos e cada um toma posse do lock de um deles, fazendo eles esperarem para sempre

16.4 Concorrência estendida com as características de *Sync* e *Send*

- Dois conceitos de concorrência estão integrados na linguagem, as características de Sync e Send
- O Send indica que a posse de valores do tipo que implementam o Send podem ser transferidos entre threads
- Quase todas as primitivas implementam essa característica, tirando ponteiros puros
- O Sync indica que é seguro que múltiplas threads tenham a referência do tipo em questão
- O Sync é o conceito mais similar de Rust para operações seguras em thread, indicando que aquele pedaço específico de dado pode ser seguramente usado por múltiplas threads concorrentes
- Implementar manualmente essas características envolve implementar código não seguro em Rust, coisas do **[Rustonomicon]**

17. O idioma *rust* comparado a princípios de *POO*.

17.1 Características de Linguagens Orientadas a objetos

- O Rust é influenciado por vários paradigmas de programação, incluindo a POO, por exemplo na seção 13 exploramos características da programação funcional
- Linguagens de POO compartilham certas características como objetos, encapsulamento e herança. Aqui vamos ver se o Rust tem suporte e como a essas características
- Podemos ter uma boa definição de POO da seguinte maneira:

"Programas orientados a objetos são feitos de objetos. Um objeto empacota tanto os dados quanto os procedimentos que operam nos dados. Os procedimentos são chamados geralmente de métodos ou operações"

- Usando esse definição, podemos dizer que o Rust é orientado a objeto: structs e enums tem dados e blocos de implementação impl
- Por mais que não sejam especificamente chamados de objetos, eles representam a funcionalidade de um

- A ideia de encapsulamento implica que os detalhes de implementação não estão acessíveis para o código que usa o objeto
- Discutimos na seção 7 como controlar o encapsulamento: podemos usar a palavra chave pub para decidir que módulos, tipos, funções e métodos serão públicos e por padrão o resto é privado
- Se o encapsulamento é um aspecto para uma linguagem ser considerada orientada a objetos, então o Rust aqui também é válido
- Herança é um mecanismo onde um objeto pode herdar elementos de definições de outros objetos, assim tendo os dados e comportamentos do pai sem ter que defini-los de novo
- O Rust não possui mecanismos de herança, ou seja de um struct herdar campos e métodos de um struct pai por exemplo sem usar uma macro
- Porém, o Rust tem outras soluções que cobrem os motivos de usar a herança em primeiro lugar
- Usamos herança por dois motivos: um deles é reutilização de código e o outro esta relacionado ao sistema de tipos, possibilitando o polimorfismo:
  - Para muitos o polimorfismo é sinônimo, mas o polimorfismo é algo bem mais geral, se referindo a códigos que funcionam com dados de vários tipos
  - O Rust usa genéricos para abstrair esse tipos diferentes e limites de características para impor restrições no que esses tipos devem providenciar
- O Rust possui mecanismos para realizar os dois motivos sem a herança, com implementações padrões de características e os conceitos de características e limites das mesmas

17.2 Usando características de objetos que possibilitam valores de diferentes tipos (*gui-structure*)

- Vamos tentar criar uma carcaça de uma biblioteca de GUI que consegue desenhar coisas padrões definidas na biblioteca e outras coisas que podem ser definidas por outros desenvolvedores
- A vantagem de usar características de objetos e o sistema de tipos de Rust é que nunca precisamos checar se um tipo implementa um método em particular em tempo de execução
- O Rust não compila nosso código se o valor não implementaras características que o objeto característico (trait object) precisa

17.3 Implementando um padrão de design orientado a objeto (*state-pattern*)

- O padrão state pattern é um orientado a objeto. Nele, definimos um conjunto de estados que um valor pode ter internamente
- Os estados são representados por objetos e o comportamento do valor muda de acordo com o estado
- Vamos fazer um exemplo para mostrar como conseguimos trabalhar com esse padrão em Rust. Implementamos apenas o primeiro exemplo

17.4 Inventário de posse #4

- Quiz para reforçar conhecimento sobre posse encontrado aqui

18. Uma referência sobre padrões e correspondência de padrões.

18.1 Todos os lugares que os padrões podem ser usados

- Padrões aparecem em uma série de lugares em Rust e usamos bastantes eles até agora
- Aqui vamos discutir todos os lugares onde esses padrões são válidos
- Usamos padrões em braços de expressões match, que formalmente podem ser definidos pela palavra chave, um valor para ser pareado com e um ou mais braços

```
match VALOR {
    PADRAO => EXPRESSAO,
    PADRAO => EXPRESSAO,
    PADRAO => EXPRESSAO,
}
```

- Um requerimento desse tipo de expressão é que ele precisa ser exaustivo, ou seja, tem que cobrir todas as possiblidades para o valor
- Temos um padrão particular em _ que pode ser pareado com qualquer coisa, mas nunca se prende a uma variável
- Expressões if let podem ser consideradas uma forma mais rápida de escrever o match que só pareia com um único caso
- Podemos misturar como quisermos if let, else if e else if let. Isso nos da grande flexibilidade
- O Rust não requer que as condições entre uma série de condições se relacionem de qualquer jeito
- A desvantagem de usar expressões if let é que o compilador não checa pela exaustão como nas expressões match, o que pode nos fazer perder algum caso
- Similar a construção do if let, o while let permite um loop condicional que continua enquanto o padrão continua a ser pareado
- Em um loop for, o valor que segue diretamente depois da palavra chave é um padrão. Por exemplo em for x in y, o x é um padrão
- Toda vez que utilizamos um let estivemos usando padrões! do tipo let PADRAO ĒXPRESSAO
- Para ver eles mais claramente, olhe os exemplos abaixo:

```
let (x, y, z) = (1, 2, 3);
// estamos pareando a tupla
// com o padrão (x, y, z).
//
// o Rust então liga o valor 1 com x,
// 2 com y e 3 com z
```

- Parâmetros de função também podem ser padrões:

```
// aqui esse x é um padrão, assim com no let
fn foo(x: i32) {
    //codigo legal
}
```

18.2 "Refutabilidade": onde padrão pode falhar a corresponder

- Padrões vem em duas formas: refutáveis e não refutáveis
- Padrões que se pareiam com qualquer valor possível passado são irrefutáveis
- Por exemplo em let x 5̄, o x pareia com qualquer coisa ou seja, não pode falhar em parear
- Padrões que podem falhar em parear com um certo valor são chamados de refutáveis

```
// exemplos de padrões refutáveis

// aqui se o valor em um_valor for None
// então o Some não vai conseguir parear
if let Some(x) = um_valor {
    //...
}

// aqui &[x, ..] é refutável pois se o um_slice
// tiver zero elementos o padrão não pode ser
// encontrado
if let &[x, ..] = um_slice {
    //...
}
```

- Parâmetros de funções, declarações let e loops for só aceitam padrões irrefutáveis, pois o programa não consegue fazer nada caso os valores não pareiem
- if let e while let permitem ambos refutáveis e não refutáveis, mas o compilador avisa contra os não refutáveis, pois por definição eles já cuidam da possível falha
- Em geral, não precisamos nos preocupar com a refutabilidade de padrões, mas precisamos estar familiarizados com o conceito

- Se temos um padrão refutável só que um padrão irrefutável é necessário podemos substituir o construto que usa o padrão ou o padrão:

```
// exemplo de padrao refutavel onde
// um irrefutavel deveria ser usado
let Some(x) = uma_opcao
// se uma_opcao for None nada pode
// ser feito e o padrao falharia
// em parear, portanto o
// compilador reclama.

// podemos arrumar isso fazendo o seguinte
if let Some(x) = uma_opcao {
    //...
}
```

- Recebemos um aviso do compilador também quando usamos uma construção que aceita ambos tipos de padrão com um que sempre vai parear:

```
if let x = 5 {
    //...
}
// aqui o 5 sempre vai se ligar ao x
// tornando o if let inutil
```

- Por essa razão, os braços de uma expressão match devem ser sempre refutáveis, menos no caso do último braço, que deve parear com qualquer valor que passe pelos outros, sendo um padrão irrefutável

18.3 Sintaxe dos padrões (*matching*)

- Aqui vamos mostrar todas as sintaxes válidas em padrões e discutir quando usar cada um
- Podemos usar o match com valores literais

```
let x = 1;

match x {
    1 => println!("Um"),
    2 => println!("Dois"),
    3 => println!("Tres"),
    _ => println!("O resto"),
}
```

- Essa sintaxe pode ser útil quando queremos que o código tome uma ação caso obtenha um dado valor concreto
- Variáveis nomeadas são padrões não refutáveis que pareiam com qualquer valor. Podemos ter complicações utilizando eles em expressões match:

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("50!");
    // esse y não é o mesmo que declaramos la em cima!!
    // o match cria um novo escopo e esse y
    // se torna um variavel nomeada que parea com tudo!
    Some(y) => println!("Pareado, y = {y}"),
    _ => println!("Caso padrão, x = {:?}", x),
}
// esse match imprime: Pareado, y = 5

println!("No fim: x = {:?}, y = {y}", x);
// esse print no final fica:
// No fim: x = Some(5), y = 10
```

- Para criar uma expressão match que compare valores x e y exteriores, teríamos que usar outra estrutura que mencionaremos mais a frente
- Podemos em um mesmo braço usar | para parear com múltiplos padrões:

```
let x = 1;

match x {
    1 | 2 => println!("um ou dois"),
    //...,
    _ =>println!("qualquer coisa"),
}
```

- Podemos usar a sintaxe ..= para dar a um braço um intervalo de valores. Só podemos usar esse intervalo com numéricos e caracteres

```
let x = 5;

match x {
    1..=5 => println!("um até cinco"),
    _ =>println!("qualquer coisa"),
}
```

- Podemos usar padrões para desconstruir structs, enums e tuplas para usar diferentes partes dos valores
- Vamos mostrar como usar no exemplo prático assim como algumas outras coisas.

Os capítulos 19 e 20 do livro são um pouco mais avançados, e não serão cobertos extensivamente nesse artigo. Os capítulos falam sobre:

19. Tópicos avançados como Rust não seguro, macros e mais sobre tempos de vida, características, tipos, funções e fechamentos.

20. Completamos um projeto onde implementamos um servidor web de baixo nível com múltiplas threads. (web-server)

Nos apêndices do livro, temos o seguinte conteúdo:

Apêndice A: Cobre palavras chave do Rust.

Apêndice B: Cobre operadores e símbolos.

Apêndice C: Cobre características deriváveis.

Apêndice D: Cobre ferramentas de desenvolvimento úteis.

Apêndice E: Explica edições do Rust.

Apêndice F: Traduções do livro.

Apêndice G: Detalha como o Rust foi feito e o que é Nightly Rust.

Capítulo 4

OpenBSD: Um sistema operacional livre focado em Segurança e Simplicidade

José Gabriel de Oliveira Santana

Acesse o artigo na íntegra clicando aqui

Resumo

OpenBSD é um sistema operacional *UNIX* livre e focado em segurança. É um sistema operacional conhecido em setores do mundo *hacker* por seu desenvolvimento focado em auditoria proativa e ênfase na simplicidade de código dos *UNIX* originais, e permitindo o grande foco em segurança dentro do código. Neste artigo, será explorado sua história, metodologia de desenvolvimento e impacto no mundo *hacker*

4.1 Introdução

OpenBSD é um sistema operacional da família dos sistemas *UNIX*. É um sistema com grande foco em segurança, isto é, em mitigação de vetores de ataque, sistemas criptográficos avançados e código simplificado com o objetivo de reduzir a quantidade de *bugs* exploitáveis. No projeto, muitos algoritmos criptográficos de estado da arte foram implementados. É importante notar que o projeto surge no Canadá - isto permite com que a exportação de tecnologias criptográficas seja possível [3]. O projeto deu origem a diversos *softwares* de rede e de segurança importantes para a produção de infraestrutura de redes - sendo o mais importante desses o *OpenSSH*, a implementação do protocolo *SSH* mais usada do mundo, disponível em todas as plataformas [6]. O sistema também tem seu foco em portabilidade. Ele está disponível para diversas arquiteturas de *hardware*, muitas inclusive descontinuadas. Isto faz com que o sistema seja muito útil para pequenos servidores ou *workstations* em dispositivos antigos, que não consigam lidar com a maioria dos sistemas operacionais e *softwares* modernos.

Figura 4.1: *Puffy*, o mascote oficial do *OpenBSD*. O peixe-espinho simboliza algo que se defende bem automaticamente.

4.2 História

Theo De Raadt, o principal desenvolvedor do *OpenBSD*, era um dos membros da equipe de desenvolvimento do NetBSD. Em 1994, ele foi convidado a renunciar o cargo dele por discordâncias

com o resto da equipe. Com isso, ele fez um *fork* da versão 1.0 do NetBSD, e começou o projeto, em 1995 [8]. O primeiro lançamento público do projeto foi em 1996, o *OpenBSD* 1.2. A partir de então, foi mantido o padrão de lançamento de uma versão nova a cada seis meses [9]. Apesar de surgir em 1995, o *OpenBSD* tem influência da história do resto dos sistemas de estilo BSD, desde 1970. O BSD era uma variante do *UNIX* desenvolvida pela Universidade de Berkeley, que tornou-se pública na versão 4.4. Esta foi a versão que deu origem ao FreeBSD, outro sistema operacional *UNIX* popular no mundo de infraestrutura.

4.3 Segurança

O *OpenBSD* possui diversas funcionalidades de segurança, algumas que inspiraram outros setores da indústria [4]:

- Alternativas a funções do padrão POSIX que estão na biblioteca padrão do C, com mitigações de segurança
- Alterações na *toolchain* de desenvolvimento
- Forte criptografia e funções de randomização, com algoritmos de estado da arte
- Sistemas de proteção de memória, impedindo acesso inválido
- Restrições de chamadas de sistema e de sistema de arquivos

A grande maioria dos programas do *OpenBSD* usa sistemas de separação de privilégios, revogação de privilégios e *chroots*, o precursor dos contêineres de hoje em dia. O *OpenBSD* foi pioneiro no uso de separação de privilégios, inspirado no princípio do menor privilégio, onde uma parte do programa roda com privilégio, e a parte que executa mais tarefas roda sem os privilégios [7]. Revogação de privilégios é semelhante à separação de privilégios - o programa roda com privilégios sobre os recursos do sistema e perde eles com o tempo. Um *chroot* é basicamente uma restrição de acesso ao sistema de arquivos. Você diz para o sistema que vai usar uma pasta qualquer no sistema de arquivos como a pasta raiz de um sistema operacional. Esse tipo de tecnologia foi a base para soluções atuais de infraestrutura como o *Docker* e *Kubernetes*, pois permite a separação de versões de *software* sem interferir no sistema *host*, sendo útil tanto como segurança e como uso de versões mais atualizadas de *software*.

Figura 4.2: *Xenodm*, o *display* manager padrão do *OpenBSD*. Possui *patches* de segurança próprios.

4.3.1 Corretude

Os desenvolvedores do projeto fazem auditorias constantes e procuram aplicar *patches* de vulnerabilidade o mais rápido possível, pois a segurança é um dos principais pontos chave do sistema. Para isso, outra meta do projeto é a corretude dos programas [3] [2]. Basicamente, o código de todo *software* que faz parte do projeto deve seguir estritamente os padrões ANSI e POSIX, e ser simples de ler e desenvolver, isto é, evitar o *overengineering* que existe em muitos *softwares* modernos [5]. Esse foco na corretude do *software* permite com que muitos *bugs* que possam levar a vulnerabilidades sejam evitados, e *patches* e auditorias sejam mais factíveis a longo prazo. O código não é obtuso, e isso permite com que seja mais prático manter ele seguro.

4.4 Desenvolvimento

O desenvolvimento do *OpenBSD* e de seus softwares inclusos é realizado pela *OpenBSD Foundation*, fundada em 2007, e por todos os colaboradores. A comunicação entre desenvolvedores

é feita através de listas de *e-mails*, um modo de comunicação "às antigas", que funciona bem para o projeto, visto que muitos dos desenvolvedores e usuários do *OpenBSD* usam sistemas mais restritos, sem muitos programas externos na parte de interface gráfica. Estão acostumados a usar mais arquivos textuais diariamente, então *e-mails* sem nenhum tipo de formatação se tornam mais eficazes.

Os *patches* enviados para serem incorporados no sistema devem seguir os ideais de corretude, e devem ser revistos com cautela. Se o contribuinte tiver histórico de boas implementações, haverá mais chances da contribuição ser aceita [1].

Softwares externos são portados para o sistema no sistema de *Ports*, por vários usuários e desenvolvedores. Nota-se que muitos desses *softwares* não passam pelo mesmo processo rigoroso de auditoria que o sistema passa, então alguns deles podem aumentar o risco de segurança e os vetores de ataque.

O projeto costuma lançar versões novas a cada 6 meses, e é um desenvolvimento contínuo. Raramente perde-se o suporte a plataformas antigas, pois muitos dos desenvolvedores mantém essas plataformas em seus laboratórios. Atualizações do projeto dificilmente causam incompatibilidades com versões antigas.

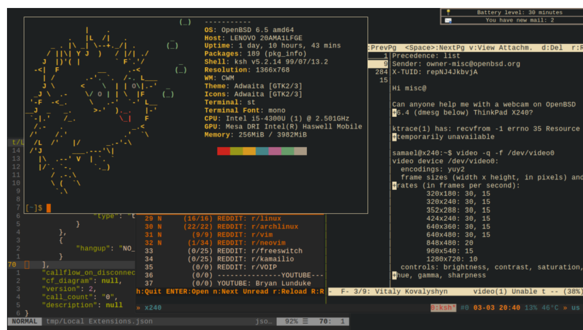

Figura 4.3: Uma *screenshot* do *OpenBSD* rodando o seu *window manager* padrão, o *cwm*. Tal como no mundo Linux, existe grande possibilidade de personalização visual do sistema.

4.5 Impacto

Várias tecnologias do *OpenBSD* foram incorporadas em outros sistemas operacionais. A mais famosa delas é o *OpenSSH*, implementação livre do protocolo *SSH*. O impacto do *OpenSSH* é um dos maiores, pois é o cliente mais usado no mundo [6]. Toda instalação *Linux* e *UNIX-like* usa o *OpenSSH*, e ele também e o cliente mais usado por padrão no *MacOS* e no *Windows*. Hoje o *OpenSSH* é integrado ao *PowerShell* e é padrão no *Windows Server*.

Outro software que é muito utilizado é o *PF*, o sistema de *firewall* do *OpenBSD*. Ele é padrão na grande maioria dos roteadores e servidores de *firewall* que existem na internet. Ele é poderoso e simples de ser configurado, tornando-se muito eficiente para uso em produção.

Como *OpenBSD* é um sistema que parece enxuto quando instalado, ele é um sistema utilizado no mundo hacker como objeto de vaidade. Usuários de *OpenBSD* aparentam ser mais avançados do que usuários de outros sistemas operacionais por estarem mais próximos do que um *UNIX* original era, visto que é o sistema que mais se assemelha a este, de todos os sistemas disponíveis hoje em dia. O poder de segurança do sistema também torna este como um dos mais interessantes para uso em servidores que estejam públicos na internet.

4.6 Licença

O *OpenBSD* usa em grande parte de seus *softwares* as licenças *BSD-2* e *ISC*, que são licenças mais permissivas do que a *GPL* e derivadas. Elas permitem que *softwares* sejam distribuídos e modificados de graça, com a única restrição de que o autor seja citado no código fonte. Diferentemente da licença *GPL*, presente no mundo do *Linux*, um *software* na licença *ISC* e *BSD-2* pode ser vendido com modificações, sem que as modificações permaneçam públicas. Isto atrai o interesse de empresas privadas que gostariam de fazer modificações ao sistema e vender as modificações.

Como essas licenças são livres, elas permiti-

ram com que os programas desenvolvidos no projeto fossem acoplados aos outros sistemas. O *OpenSSH* pode ser exportado para diversas plataformas por conta desse licenciamento.

4.7 Conclusão

Com base no que foi explorado, pode-se ver que o *OpenBSD* tem um espaço importante no mundo da tecnologia. Foi pioneiro em sistemas de segurança avançados e implementação de algoritmos criptográficos importantes. Seu caráter simples e de "volta às raízes"o torna um sistema interessante para *hackers* que gostam de trabalhar com recursos limitados e que precisam que seus sistemas sejam robustos. Também são um interessante ponto de entrada para o desenvolvimento de sistemas operacionais, se puder fazer boas contribuições, além de permitir com que tenha uma boa visão da história dos sistemas operacionais mais usados em infraestrutura de rede hoje em dia.

4.8 Bibliografia

[1] Jeremy andrews. interview: Theo de raadt. URL: `https://web.archive.org/web/20130424125958/http://kerneltrap.org/node/6550`.

[2] Federico biancuzzi. an interview with openbsd's marc espie. URL: `https://web.archive.org/web/20180504070533/http://www.onlamp.com/pub/a/bsd/2004/03/18/marc_espie.html`.

[3] Faq - introduction to openbsd. URL: `https://www.openbsd.org/faq/faq1.html`.

[4] Innovations. URL: `https://www.openbsd.org/innovations.html`.

[5] Openbsd from a veteran linux user perspective. URL: `https://cfenollosa.com/blog/openbsd-from-a-veteran-linux-user-perspective.html`.

[6] Privilege separated openssh. URL: `http://www.openssh.com/usage/`.

[7] Niels provos. privilege separated openssh. URL: `https://web.archive.org/web/20120102075206/http://www.citi.umich.edu/u/provos/ssh/privsep.html`.

[8] Theo de raadt. archive of the mail conversation leading to theo de raadt's departure. URL: `http://www.theos.com/deraadt/coremail.html`.

[9] Theo de raadt. the openbsd 2.0 release. URL: `http://wolfram.schneider.org/bsd/ftp/releases/OpenBSD-2.0`.

Capítulo 5

Compressão: O Que Torna a Tecnologia Moderna Viável

Ryan Guerra Sakurai

OBS: Para ver o artigo na íntegra, com imagens maiores, clique **aqui.**

5.1 Introdução

Em um mundo cada vez mais dependente da tecnologia, arquivos digitais são a espinha dorsal da informação que circula pela *internet*. No entanto, apesar de parecerem mágicos, assim como computadores no geral, arquivos são apenas uma sequência de número representados por *bits* (dígitos binários). Esses *bits* são armazenados de maneira sequencial, precedidos por um cabeçalho repleto de informações, também representadas numericamente, que dão significado a essa sequência de *bits* e a identificam como um arquivo. Esta composição, essencialmente simples, tem um papel fundamental na nossa vida digital.

A necessidade de transferir arquivos pela *internet* de forma rápida e eficiente e o constante desejo de economizar espaço de armazenamento impulsionaram uma área crucial da ciência da computação: a compressão de dados. Compressão, como o próprio nome sugere, é o processo de reduzir o tamanho de um arquivo, ou seja, reduzir a quantidade de números necessários para o representar. Esta prática tem se tornado cada vez mais essencial em nosso cotidiano digital, tornando possível a transmissão veloz de dados e a otimização dos recursos de armazenamento.

Neste artigo, exploraremos o mundo da compressão de arquivos, explorando os tipos e métodos de compressão e como eles são usados no dia a dia. Afinal, compreender a compressão é compreender a eficiência por trás do funcionamento do nosso mundo digital.

5.2 Tipos de Compressão

A compressão de dados pode ser feita de diversas maneiras diferentes. Para alcançar esse objetivo, existem duas abordagens fundamentais, cada uma com suas características únicas e aplicações específicas. Nesta seção, entenderemos e diferenciaremos os dois principais tipos de compressão: *lossless compression* (compressão sem perda) e *lossy compression* (compressão com perda). Cada uma dessas abordagens desempenha um papel crucial no mundo da tecnologia da informação, moldando a maneira como armazenamos, transmitimos e utilizamos os dados digitais.

Lossless Compression

A compressão sem perdas é um método que busca reduzir o tamanho de um arquivo sem comprometer a qualidade ou a integridade dos dados. Em outras palavras, após a compressão e a subsequente descompressão, o arquivo recuperado será idêntico ao original. Isso é fundamental

em situações em que a exatidão dos dados é crítica, como em arquivos de texto e códigos fonte, apesar de também ser usado em outros tipos de arquivo.

Este tipo de compressão funciona identificando padrões repetitivos ou redundâncias nos dados e substituindo esses padrões por representações mais compactas. Um paralelo a isso seria transformar a conta matemática $5 * 5 * 5 * 5 * 5 * 5$ em 5^6.

Lossy Compression

Em contraponto, a compressão com perdas é uma técnica que prioriza a redução significativa do tamanho do arquivo em troca de uma perda controlada de qualidade. Esse tipo de compressão é amplamente empregado em mídias como imagens, áudio e vídeo e muita vezes abusado em serviços de *streaming* como *Youtube* e *Twitch*, onde há necessidade de economia de largura de banda. Isso se dá devido ao fato de que, nesses caso, pequenas perdas na qualidade podem ser aceitáveis em troca de tamanhos de arquivo muito menores.

Em compressão com perdas, as informações consideradas menos importantes ou imperceptíveis para os sentidos humanos são descartadas ou representadas de forma mais compacta. Para isso ser realizado com sucesso são usados diversos conceitos da psicofísica, uma área da psicologia que estuda estímulos físicos e as respostas psicológicas e sensações que eles produzem.

5.3 Compressão de Propósito Geral

DEFLATE

Um dos algoritmos mais amplamente adotados para alcançar a compressão sem perdas é o algoritmo *Deflate*. Este algoritmo é uma peça fundamental nos formatos de arquivo *.zip*, amplamente utilizado no ambiente *Windows*, e *.gz*, uma escolha comum em sistemas *Linux*. O algoritmo *Deflate* combina dois algoritmos: o *Huffman Coding* e o LZSS (*Lempel-Ziv-Storer-Szymanski*), descendente do popular LZ77. Nesta seção, exploraremos em detalhes como o algoritmo *Deflate* opera, explicando os dois algoritmos que o formam.

Huffman Coding

Huffman Coding é uma técnica de *prefix coding* (codificação de prefixo), ou seja, nenhum código gerado por ela é prefixo de outro código. Por exemplo, se o código 001 foi gerado, não existirá nenhum outro código que comece com 001. Esta técnica opera com base no princípio de atribuir códigos de comprimento variável a cada caractere (representado por um *byte*) com base em sua frequência de ocorrência nos dados a serem comprimidos.

O processo de codificação *Huffman* começa atribuindo pesos a cada caractere, representando suas frequências relativas. Os dois elementos com pesos mais baixos são selecionados e combinados em nós de uma árvore binária, onde o ramo esquerdo é rotulado com 0 e o direito com 1. Esse processo continua até que todos os elementos estejam incorporados à árvore, resultando em uma estrutura hierárquica conhecida como "árvore de *Huffman*". Com isso, para encontrar o código de um determinado caractere, basta concatenar os *bits* equivalentes ao caminho até seu nó.

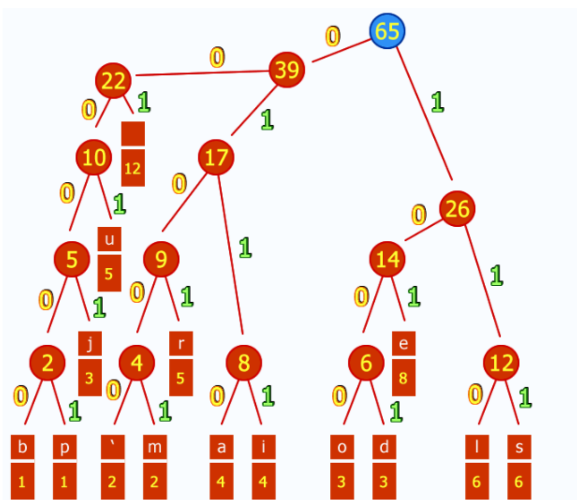

Figura 5.1: Figura ilustrando a Construção da árvore de *Huffman*. clique aqui para ver uma versão animada em *gif*.

Construção da árvore de *Huffman*

A utilidade da codificação *Huffman* para a compressão está na sua capacidade de gerar códigos mais curtos para elementos que ocorrem com frequência e códigos mais longos para aqueles menos comuns, o que efetivamente reduz o tamanho geral dos dados codificados. Além disso, a árvore de *Huffman* em si pode ser usada para decodificar os dados comprimidos de maneira eficiente.

LZSS

O algoritmo LZSS é uma evolução do LZ77, que se enquadra na categoria das técnicas de *dictionary coding*. O LZSS é um algoritmo de compressão que se baseia na ideia de manter constantemente um registro de uma quantidade pré-determinada de caracteres anteriores, conhecido como "janela deslizante".

A principal funcionalidade do LZSS é identificar sequências de caracteres que são idênticas a aquelas já presentes na janela deslizante. Quando uma correspondência é encontrada, a sequência repetida é substituída por uma referência a essa ocorrência. Essa referência é composta por dois elementos essenciais: a distância entre a sequência duplicada e a original dentro da janela deslizante, e a quantidade de caracteres duplicados.

É importante destacar que essa parte do algoritmo, ou seja, a busca por correspondências e a criação de referências, é a parte mais computacionalmente cara do DEFLATE e, como resultado, a otimização dessa parte do algoritmo é crítica para a eficiência geral da compressão.

Devido à complexidade e à importância da implementação dessa funcionalidade, a forma como o LZSS é implementado pode variar consideravelmente de uma implementação para outra. Diferentes implementações podem utilizar estratégias diferentes para otimizar o processo de busca por correspondências e a criação de referências, o que pode resultar em desempenho e eficiência variados. Portanto, essa é a parte do algoritmo DEFLATE que está mais sujeita a alterações de acordo com a implementação específica e os objetivos de compressão desejados.

0: I meant what I said	0: I meant what I said
20: and I said what I meant	20: and(12,7)(7,8)(2,5)
44:	30:
45: From there to here	31: From there to (51,4)
64: from here to there	47: f(46,4)(51,8)(50,5)
83: I said what I meant	55: (24,19)
Total characters 102	Total characters 56

Figura 5.2: Figura ilustrando a codificação pelo algoritmo LZSS. Clique aqui para ver a imagem ampliada.

Codificação por LZSS

5.4 Compressão de Imagem

PNG

O formato de imagem PNG, amplamente conhecido por sua capacidade de suportar transparência em imagens, é outro exemplo em que há uso de *lossless compression*. Inclusive, assim como ZIP e GZIP, este formato também faz uso do algoritmo DEFLATE, que é aplicado em cada canal de cor da imagem (vermelho, verde e azul). No entanto, a aplicação dessa técnica em imagens é um processo complexo devido à natureza visual dos dados, que não apresenta redundâncias tão evidentes quanto em texto.

Para superar esse desafio, o PNG emprega uma abordagem peculiar. Além do uso do algoritmo de compressão DEFLATE, mencionado na seção anterior, o PNG incorpora um mecanismo adicional. Esse mecanismo envolve a aplicação de um filtro que o algoritmo considera apropriado em cada canal de cor, para destacar padrões na imagem e, assim, forçar a criação de redundâncias artificiais. A ideia por trás disso é tornar os dados da imagem mais suscetíveis à compressão eficiente pelo algoritmo DEFLATE.

Em outras palavras, o filtro aplicado durante a compressão do PNG visa criar uma versão modificada da imagem que contenha mais infor-

mações sobre a estrutura e os padrões da imagem, tornando-a mais comprimível. É importante notar que a escolha do filtro é determinada pelo algoritmo do PNG com base em uma heurística, ou seja, uma estimativa aplicada na imagem em questão.

No entanto, embora a abordagem do PNG para compressão de imagens seja eficaz em termos de preservação da qualidade, ela também tem suas desvantagens. Assim como o algoritmo DEFLATE, os processos de compressão e descompressão do PNG tendem a ser relativamente lentos, especialmente em comparação com outros formatos de imagem mais rápidos, como o JPEG. Portanto, a escolha do formato de imagem a ser utilizado depende das necessidades específicas do usuário em relação à qualidade, tamanho do arquivo e desempenho.

JPEG

No mundo da compressão de imagens, o JPEG é, provavelmente, o formato mais renomado. Sua popularidade se deve, em grande parte, à sua eficiência em reduzir o tamanho de arquivos de imagem sem comprometer significativamente a qualidade visual. O segredo por trás da eficácia do JPEG reside em sua capacidade de explorar as nuances da nossa visão.

Nossos olhos são órgãos altamente sensíveis à luz, e, portanto, somos muito mais perceptíveis às variações no nível de iluminação (luminância) em uma imagem do que às variações de cor (crominância). O JPEG inteligentemente usa essa característica da nossa percepção visual para criar imagens compactas por meio de cinco etapas: *Color space conversion*, *Chrominance Downsampling*, *Discrete Cosine Transform*, Quantização e *Run-Length Encoding & Huffman Coding*.

Etapa 1. *Color Space Conversion*

A primeira etapa da compressão é a conversão do espaço de cores (*color space conversion*). Cada *pixel* de uma imagem é composto por três elementos: o nível de vermelho (R), o nível de verde (G) e o nível de azul (B), que juntos determinam a cor de cada *pixel*. Nesta etapa, esses três elementos de cor são utilizados para encontrar três novos valores: luminância (Y), crominância vermelha (Cr), e crominância azul (Cb).

O resultado dessa etapa é que os *pixels* da imagem são agora representados por esses três novos elementos (Y, Cb e Cr) em vez dos valores originais de vermelho, verde e azul. O importante a notar é que os *pixels* formados com esses novos elementos são equivalentes em termos de informação visual. Isso significa que a conversão é reversível, e a imagem pode ser restaurada para o seu espaço de cores original sem perda de informações.

A conversão para o espaço de cores *YCbCr* é crucial no processo de compressão JPEG, uma vez que permite que as informações de crominância (Cb e Cr) sejam quantificadas de forma mais agressiva do que a luminância (Y). Isso é possível porque nossos olhos são menos sensíveis a variações de cor em comparação com variações de brilho. Portanto, ao explorar essa característica da visão humana, o JPEG pode alcançar uma alta taxa de compressão sem comprometer gravemente a qualidade visual da imagem.

Etapa 2. *Chrominance Downsampling*

Nesta etapa, os componentes Cr e Cb são divididos em blocos de 2x2 *pixels*, o que significa que cada bloco contém quatro valores de cada componente. Para cada bloco, é calculada a média dos valores dos *pixels* de Cr e a média dos valores dos *pixels* de Cb. Isso resulta em dois novos valores, um para Cr e outro para Cb, que representam a crominância média para esse bloco específico.

Cada *pixel* dentro do bloco é substituído por essa média correspondente. Portanto, todos os *pixels* no mesmo bloco agora terão o mesmo valor de crominância. O resultado dessa etapa é que, em comparação com a imagem original, dois dos três componentes de crominância (Cr e Cb) agora estão com apenas 1/4 das informações originais, uma vez que a média é compartilhada entre os quatro *pixels*.

Essa redução na resolução da crominância não é geralmente perceptível para o olho humano, uma vez que a crominância é menos sensível à resolução do que a luminância. No entanto, essa etapa desempenha um papel fundamental na compressão eficaz de imagens JPEG.

Etapas 3 e 4. *Discrete Cosine Transform* & Quantização

Os processos de *Discrete Cosine Transform* (DCT) e quantização funcionam baseadas no fato de que nossos olhos não são eficazes em perceber detalhes de alta frequência em uma imagem. Por exemplo, podemos claramente distinguir os limites dos troncos das árvores, mas não conseguimos perceber com clareza cada grama no chão ou cada folha nas árvores nesta imagem:

Figura 5.3: Foto de uma floresta (autoria de *Felix Mittermeier*). Clique aqui para ver a imagem ampliada.

Para abordar essa limitação perceptual, o processo de DCT e quantização varre todas as três componentes de imagem (Y, Cr e Cb) e identifica elementos com alta frequência. Em seguida, esses elementos de alta frequência são removidos ou reduzidos. Isso é feito por meio de um processo matemático extremamente complexo.

É feita uma transformação matemática chamada DCT, que converte os *pixels* em coeficientes que representam quanta informação de alta frequência está presente em uma determinada região da imagem. Após isso, o processo de quantização é aplicado, onde esses coeficientes são arredondados para níveis específicos, o que resulta em perda de informações de alta frequência. A quantização é responsável por criar redundâncias na imagem, uma vez que vários coeficientes podem ser quantizados para o mesmo valor. Essa redundância é explorada na próxima etapa de codificação.

Em ferramentas como *Adobe Photoshop*, é possível escolher o nível de qualidade de uma foto JPEG. Neste caso, quanto menor a qualidade escolhida for, mais agressivo será este processo. Isso fará com que o tamanho do arquivo seja ainda menor, mas em casos em que a qualidade for muito baixa, falhas chamadas de “artefatos” ficam visíveis.

Etapa 4. *Run-Length Encoding & Huffman Coding*

O *Run-Length Encoding* (RLE) é um método de compressão que se assemelha ao mencionado LZSS, embora seja consideravelmente mais simples em sua abordagem. Sua operação consiste em abreviar sequências de caracteres ou *bytes* repetidos, representando-os apenas uma vez, seguidos pelo número de ocorrências. Por exemplo, a sequência "AAAAAABCCCCC" seria reduzida a algo similar a "A[x6]BC[x5]". Dado que a etapa anterior do processo JPEG frequentemente gera valores repetidos, o RLE se torna altamente eficaz nesse contexto. Após isso, é usado *Huffman Coding*, que foi previamente explicado neste artigo.

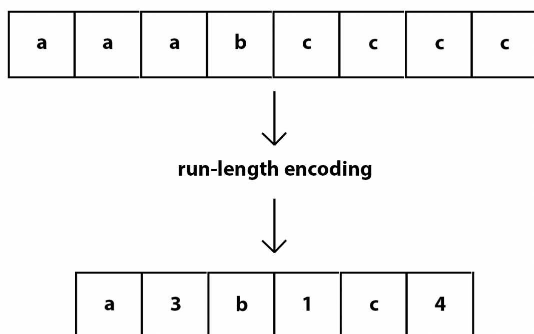

Figura 5.4: Ilustração do processo de RLE. Clique aqui para ver a imagem ampliada.

Ilustração do processo de RLE

5.5 Compressão de Áudio

MP3

Assim como no formato JPEG para imagens, o formato MP3 também explora as limitações dos sentidos humanos em sua vantagem. Isso é alcançado por meio do uso de conhecimentos em modelagem psicoacústica e codificação perceptual.

A modelagem psicoacústica no MP3 envolve a compreensão das características da percepção auditiva humana. O algoritmo é projetado para descartar sons em frequências que os seres humanos não conseguem ouvir ou que são menos perceptíveis. Quando ouvimos um som alto, isso pode fazer com que sons em frequências semelhantes se tornem menos perceptíveis devido a um fenômeno conhecido como mascaramento auditivo. Portanto, o algoritmo representa esses sons "vizinhos"em menor fidelidade, economizando espaço.

Um aspecto flexível do MP3 é a capacidade de obter resultados com diferentes tamanhos e qualidades de áudio usando o mesmo algoritmo. Isso é alcançado por meio do ajuste do *bit rate*. Quanto maior o *bit rate* selecionado, maior será a qualidade do áudio resultante, pois mais dados são alocados para representar as nuances do som. Por outro lado, ao reduzir o *bit rate*, o arquivo de áudio é comprimido com mais força, resultando em menor qualidade, mas em um tamanho de arquivo menor.

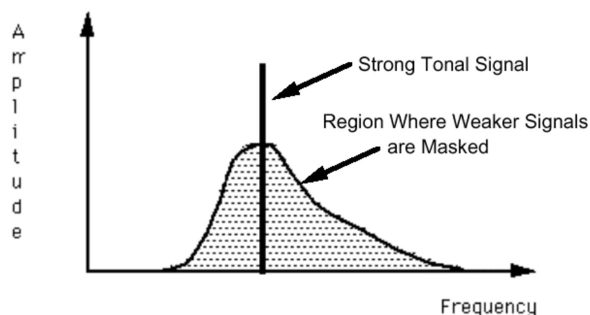

Figura 5.5: Ilustração do fenômeno de mascaramento auditivo. Clique aqui para ver a imagem ampliada.

Ilustração do fenômeno de mascaramento auditivo

5.6 Compressão de Vídeo

Ao contrário das outras formas de mídia digital citadas anteriormente, como imagens estáticas ou arquivos de áudio, o formato de um vídeo não é determinado apenas pelo seu tipo. Em vez disso, ele é definido por dois componentes: o *codec* e o *container*.

O *codec*, um termo que muitas vezes permanece desconhecido para a maioria dos usuários casuais de computadores, é um algoritmo especializado responsável por codificar e decodificar o vídeo. É este *codec* que nos interessa nesta seção, já que é o elemento central na compressão de vídeo. Enquanto o formato do vídeo é moldado por esse algoritmo, o *container* é utilizado para manter todos os dados relativos ao vídeo em um único arquivo. Isso inclui não apenas o vídeo propriamente dito, mas também o áudio, metadados, legendas e outros elementos associados.

Exemplos comuns de *containers* incluem MP4, AVI, MOV e MKV. No entanto, entre todos os *co-*

decs existentes, o H.264, também conhecido como AVC (*Advanced Video Coding*), destaca-se como o mais amplamente utilizado na atualidade. Portanto, nesta seção, focaremos nosso estudo no algoritmo de compressão do *codec* H.264.

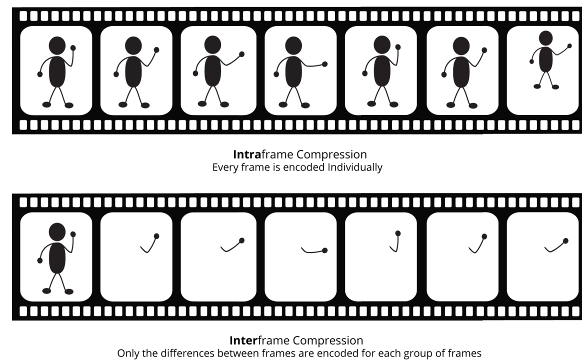

Figura 5.6: Ilustração dos dois tipos de compressão de vídeo. Clique aqui para ver a imagem ampliada.

Ilustração dos dois tipos de compressão de vídeo

H.264

O H.264, conhecido por sua versatilidade e eficiência, é um *codec* que aborda ambas as formas de de compressão de vídeo: a compressão espacial (*intraframe*) e a compressão temporal (*interframe*). A compressão espacial, em essência, assemelha-se à compressão de imagem, reduzindo a redundância dentro de cada quadro (*frame*) do vídeo. Vai além dos processos tradicionais utilizados pelo algoritmo JPEG, gerando uma imagem com base na predição de *pixels* a partir dos *pixels* na mesma imagem. Essa imagem predita é, então, subtraída da imagem real, resultando no que chamamos de "resíduo". A vantagem aqui está na transmissão das informações de como a predição foi feita e o resíduo, o que resulta em economia significativa de espaço.

A compressão temporal, por outro lado, tira proveito do fato de que a maioria das sequências de vídeo consiste em *frames* consecutivos que são quase idênticos na maior parte do tempo. Nesses casos, não é necessário codificar cada *frame* individualmente. Em vez disso, apenas as diferenças entre esses *frames*, como movimentos, são codificadas. No contexto do H.264, um vídeo é composto por três tipos principais de *frames*: *i-frames*, *p-frames* e *b-frames*. Os *i-frames* são essencialmente imagens JPEG independentes. Os *p-frames* contêm as mudanças feitas desde o último *i-frame* ou *p-frame*, enquanto os *b-frames* são interpolados entre *i-frames* e *p-frames*, aproveitando ao máximo a redundância temporal.

5.7 Conclusão

Em um mundo cada vez mais dependente da tecnologia, a compressão de dados se revelou uma peça fundamental no nosso cotidiano digital. Este artigo explorou os diversos aspectos da compressão, desde os tipos de compressão, como a *lossless* e a *lossy*, até os algoritmos e técnicas específicos usados em diferentes tipos de mídia, como imagens, áudio e vídeo.

Assim, este artigo nos trouxe uma visão mais profunda da compressão de dados e sua onipresença em nosso dia a dia digital. A compreensão dessas técnicas e algoritmos nos permite apreciar a eficiência por trás do funcionamento do nosso mundo digital e como a compressão de dados continua a moldar a forma como interagimos e compartilhamos informações na era da tecnologia.

5.8 Bibliografia

[1] Files & file systems: Crash course computer science #20. URL: https://youtu.be/KN8YgJnShPM?si=wramOz88Ze3ouHA7.

[2] Compression: Crash course computer science #21. URL: https://youtu.be/OtDxDvCpPL4?si=rsqwe3gxuOF7BHsy.

[3] Data compression as fast as possible. URL: https://youtu.be/guo8if4Yxhw?si=xuss4eJvYVzHmF8h.

[4] Data compression. URL: https://en.wikipedia.org/wiki/Data_compression.

[5] Video compression as fast as possible. URL: https://youtu.be/qbGQBT2Vwvc?si=CnvaeF75EprpGvCN.

[6] Explaining digital video: Formats, codecs & containers. URL: https://youtu.be/-4NXxY4maYc?si=QUPLwpmFm2iJCl3c.

[7] Audio file formats - mp3, aac, wav, flac. URL: https://youtu.be/WIIKXOrt3bk?si=Bxe83WNai3EB2OOr.

[8] An explanation of the 'deflate' algorithm. URL: https://www.zlib.net/feldspar.html.

[9] Lempel–ziv–storer–szymanski. URL: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski.

[10] How png works: Compromising speed for quality. URL: https://youtu.be/EFUYNoFRHQI?si=gR0rd4r2YreVF72_.

[11] How are images compressed? [46mb -> 4.07mb] jpeg in depth. URL: https://youtu.be/Kv1Hiv3ox8I?si=DyVMCU8UQvGG7A6T.

[12] Run-length encoding. URL: https://en.wikipedia.org/wiki/Run-length_encoding.

[13] How mp3 file works | mp3 compression explained in 3 minutes. URL: https://youtu.be/Jfl8ldZkxc0?si=0w3KE1a_PpBztau8.

[14] Digital audio compression - computerphile. URL: https://youtu.be/KGZ0een8vSE?si=1bF6itWpiiyZBiCm.

[15] Mp3: Concept. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/mp3/concept.htm.

[16] Mp3: Psychoacoustic model. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/mp3/psychoacoustics.htm).

[17] Video compression as fast as possible. URL: https://youtu.be/qbGQBT2Vwvc?si=qO4F_dIIQFx2efQF.

[18] Understanding video file formats, codecs and containers. URL: https://www.techsmith.com/blog/video-file-formats/.

[19] Difference between inter and intra frame compression - geeksforgeeks. URL: https://www.geeksforgeeks.org/difference-between-inter-and-intra-frame-compression/.

[20] Explaining digital video: Formats, codecs & containers. URL: https://youtu.be/-4NXxY4maYc?si=5mNFumWbOkC6ekqN.

[21] Learn h.264/avc in 3 minutes with playclaw setup. URL: https://youtu.be/0SXa8iQZMjo?si=J3tvbRTq1rQaferR.

Capítulo 6

Decodificando as Linguagens de Programação

Ryan Guerra Sakurai

OBS: Para ver o artigo na íntegra, com imagens maiores, clique **aqui.**

As linguagens de programação são ferramentas que capacitam os programadores a construir programas de computador, indo desde os mais simples, como calculadoras, aos mais complexos, como agentes de inteligência artificial. Analogamente, elas funcionam como uma espécie de língua, permitindo que os seres humanos comuniquem instruções aos computadores e solicitem a execução de tarefas específicas. Este diálogo entre humanos e máquinas é facilitado por diferentes tipos de linguagens, divididas entre aquelas de baixo nível, mais próximas do código binário utilizado pelas máquinas, e as de alto nível, que se assemelham mais às línguas faladas pelos humanos.

Nos primórdios da computação, os programadores trabalhavam predominantemente com linguagens de baixo nível, enquanto, atualmente, a esmagadora maioria deles opta por linguagens de alto nível. Por isso, é sobre essas linguagens que este artigo discorrerá, classificando-as de acordo com suas características, apresentando os paradigmas de programação que influenciam as abordagens de desenvolvimento e oferecendo exemplos concretos de linguagens usadas atualmente para construir ferramentas usadas no nosso dia-a-dia.

6.1 Compilação e Interpretação

Embora comumente rotulemos linguagens como "compiladas"ou "interpretadas", tecnicamente, essa classificação é imprecisa. A natureza compilada ou interpretada de uma linguagem é uma característica da implementação, não da linguagem em si. Na prática, é perfeitamente possível criar um compilador para uma linguagem geralmente considerada interpretada ou desenvolver um interpretador para uma linguagem geralmente vista como compilada.

Mesmo ao desconsiderarmos essa tecnicidade, o título de uma linguagem como "interpretada"ou "compilada"permanece obscuro, com controvérsias em relação a exemplos práticos. Essa ambiguidade é atribuída, em parte, ao fato de que muitas linguagens, especialmente aquelas rotuladas como interpretadas, frequentemente adotam abordagens mistas. Por isso, serão tratadas definições teóricas nesta seção e exemplos particulares mais a frente no artigo.

É considerado que em uma linguagem compilada, o desenvolvedor escreve o código em uma linguagem de programação, e todo o código é convertido integralmente em linguagem de máquina durante o processo de compilação. Esse resultado é um executável independente que pode ser executado sem a necessidade do código-fonte original. Esse método proporciona uma execução rápida, pois a tradução para código de máquina ocorre durante a compilação, otimizando completamente o código para o sistema alvo.

Em contrapartida, linguagens interpretadas não geram um executável. Cada linha do código é convertida em código de máquina em tempo real por um interpretador, que é necessário durante toda a execução do programa. Embora a execução seja mais lenta em comparação com linguagens compiladas, a vantagem está na ausência de uma etapa de compilação, permitindo a execução imediata do código assim que é escrito.

A compilação pode ser um processo demorado, mas oferece benefícios, como a total otimização do código, a ausência da responsabilidade de conversão do código durante execução e a privacidade do código-fonte, já que apenas o executável é distribuído. Por outro lado, em linguagens interpretadas, o próprio código-fonte é executado, proporcionando facilidade de distribuição, mas expondo o código ao ambiente de execução.

No contexto de linguagens compiladas, o executável resultante não é multiplataforma, requerendo recompilação para cada sistema em que será executado. Em contraste, linguagens interpretadas podem ser executadas em qualquer sistema que possua o interpretador correspondente. Contudo, a execução nesse caso fica dependente do interpretador, o que pode ser considerado uma desvantagem.

- Compilação

- Interpretação

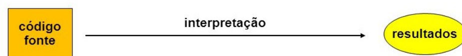

Figura 6.1: Figura ilustrando os processos de Compilação vs Interpretação. Clique aqui para ver a imagem ampliada.

Compilação vs Interpretação

6.2 Sistemas de Tipos

O sistema de tipos de uma linguagem é um conjunto de regras em que cada valor ou dado em um programa tem um tipo, sendo que os possíveis tipos variam de acordo com a linguagem. Esses são alguns exemplos de valores e seus tipos:

```
12 //(número inteiro)
12.25 //(número de ponto flutuante)
"pirata que estica" //(cadeia de caracteres).
```

O sistema de tipos de uma linguagem é uma de suas características essenciais, exercendo influência significativa na manipulação de dados durante a programação. Nesse contexto, existem duas distinções fundamentais entre linguagens: a tipagem estática em contraste com a tipagem dinâmica, e a tipagem forte comparada à tipagem fraca. Contudo, é interessante observar que não é raro encontrar, por exemplo, linguagens que, apesar de serem predominantemente de tipagem forte, apresentam algumas características de linguagens de tipagem fraca, evidenciando a complexidade do assunto.

Tipagem Estática e Tipagem Dinâmica

Em geral, em linguagens com tipagem estática, o tipo de uma variável é determinado durante o tempo de compilação e permanece constante ao longo do código. Essas linguagens são comumente compiladas, pois essa abordagem é eficaz para lidar com a tipagem estática.

Por outro lado, em linguagens com tipagem dinâmica, o tipo de uma variável só é conhecido durante a execução do programa, podendo variar dependendo do ponto em que a execução está sendo analisada. Linguagens com essa característica geralmente são interpretadas, pois a tipagem dinâmica e a interpretação se complementam.

Em outras palavras, o que diferencia esses dois grupos é o momento em que acontecem as checagens de tipo. Em linguagens dinâmicas, ela acontece em tempo de execução, enquanto em linguagens estáticas ela acontece em tempo de compilação, antes da execução.

Ao comparar os dois estilos de tipagem, as vantagens de um geralmente coincidem com as desvantagens do outro. Linguagens dinâmicas oferecem maior flexibilidade, dinamicidade e exigem menos código, porém, em troca, podem sacrificar robustez, previsibilidade, desempenho e proteção contra *bugs*, características presentes em linguagens estáticas.

Por outro lado, linguagens estáticas são conhecidas por sua robustez, segurança, maior desempenho e previsibilidade. No entanto, podem ser consideradas mais rígidas e verbosas em comparação com as linguagens dinâmicas.

```
// Tipagem Estática:

Integer x = 420;
x = "Pinhalzinho - SP"
// gera erro de compilação
```

```
# Tipagem Dinâmica:

x = 69
x = "antes sofria, hoje sou fria"
# permitido
```

Tipagem Forte e Tipagem Fraca

Na subseção anterior, delineamos a distinção entre linguagens com base no momento da realização da checagem de tipos. Agora, voltamos nossa atenção para a rigidez dessa checagem, sendo que a natureza desta diferenciação é menos clara, uma vez que se trata de um espectro contínuo.

A tipagem de uma linguagem, quando mais fraca, amplia a probabilidade de ocorrerem conversões implícitas de tipo. Em outras palavras, a frequência dessas conversões está intrinsecamente ligada ao grau de permissividade concedido pela linguagem na manipulação de tipos de dados. Como mostrado a seguir, ainda mais que a tipagem dinâmica, uma tipagem mais fraca pode comprometer a segurança e a previsibilidade de uma linguagem:

```
// Característica de Tipagem Forte:

int a = 10;
String b = "2";
a = a + b;  // Gera erro de tipo
```

```
// Característica de Tipagem Fraca:

let a = 10
const b = "2"
a = a + b   // a = "102"
```

6.3 Paradigmas de Programação

Paradigmas de programação referem-se a maneiras de estruturar e conceber código, essencialmente representando abordagens distintas para resolver problemas por meio da programação. São, portanto, maneiras de programar.

A categorização dos paradigmas divide-se principalmente em dois tipos: programação imperativa e programação declarativa. A programação imperativa envolve a explicitação de um passo a passo de ações a serem executadas, baseando-se na mudança de estado do programa e no controle do fluxo de execução, refletindo de perto o funcionamento interno de um computador.

Por outro lado, a programação declarativa concentra-se apenas na declaração do problema e na especificação do resultado desejado, sem prescrever a sequência exata de passos para atingir esse resultado. Essa abordagem é mais abstrata, distanciando-se do nível de detalhes da implementação.

Linguagens de programação podem adotar exclusivamente um paradigma, mas é mais comum que uma linguagem ofereça suporte a vários paradigmas, mesmo que um deles seja predominante. Nesta seção, apresentaremos os paradigmas mais amplamente utilizados: programação procedural, programação orientada a objetos e programação funcional.

Programação Procedural

A programação procedural, pertencente à categoria de programação imperativa, é caracterizada pela estruturação do programa em procedimentos. Estes procedimentos consistem em trechos distintos de instruções que se se executam entre si para alcançar um objetivo específico.

A estruturação por procedimentos confere à programação procedural a vantagem da reusabilidade do código, uma vez que um mesmo procedimento pode ser invocado em diferentes partes do programa. Esta característica promove a modularidade e facilita a manutenção do código, por torná-lo menor e mais legível. Além disso, a atualização de um procedimento reflete automaticamente em todas as chamadas realizadas ao longo do código.

A programação procedural apresenta o menor nível de abstração entre os três paradigmas sendo apresentados. Em virtude de sua simplicidade e proximidade com o funcionamento interno de um computador, a programação procedural é mais amplamente adotada por linguagens de mais baixo nível, onde é exigido maior controle sobre o hardware e o desempenho é crítico.

```
// Exemplo de uso do paradigma procedural:

float somar(float a, float b) {
    return a + b;
}

float calcular_media(float numeros[]
    , int qt_numeros) {
    float soma = 0;
    for(int i=0; i < qt_numeros; i = i+=1)
        soma = somar(soma, numeros[i]);
    return soma / qt_numeros;
}

int main() {
    float nums[] = {1, 2, 3, 4};
    float media = calcular_media(nums, 4);

    return 0;
}
```

Programação Orientada a Objetos

A programação orientada a objetos (POO) é um paradigma imperativo amplamente adotado na atualidade, destacando-se como o mais popular entre os desenvolvedores. Nesse paradigma, um programa é estruturado em torno de objetos, entidades que possuem tanto atributos (dados) quanto métodos (comportamentos).

Em grande parte das linguagens de programação, os objetos são definidos por meio de classes. As classes são essencialmente descrições de quais características um objeto deve possuir e quais ações ele deve ser capaz de realizar. Os objetos são então instâncias concretas dessas classes, moldados conforme as especificações da classe.

Para ilustrar esse conceito, consideremos uma classe que define que um cachorro possui os atributos raça e nome, juntamente com métodos para latir e correr. Podemos criar diferentes instâncias dessa classe, como um cachorro da raça *Bulldog* Francês chamado Sushi e outro da raça *American Bully* chamado Zara. Além disso, podemos fazer os dois cachorros de fato latirem e até interagir um com outro (por exemplo, um correr atrás do outro).

Além disso, o paradigma incorpora conceitos poderosos como a herança, que possibilita o reaproveitamento de código ao permitir que uma classe herde características e comportamentos de outra. Reutilizando o exemplo anterior, podemos ter uma classe Cachorro como base e criar subclasses específicas como *Bulldog* Francês e *American Bully*, herdando as características da classe principal. Isso proporciona uma organização hierárquica que reflete a relação entre diferentes tipos de cachorros.

O polimorfismo é outra característica relevante, permitindo que objetos de diferentes classes sejam tratados de maneira uniforme. No exemplo dos cachorros, podemos ter uma função que aceita qualquer instância de cachorro como parâmetro, independentemente da raça específica. Isso aumenta a flexibilidade e reutilização do código.

A POO é considerada a melhor maneira de modelar o mundo por meio de código. Essa metodologia proporciona não apenas reusabilidade de código e modularidade, características já presentes na programação procedural, mas também introduz abstração e encapsulamento. A comunicação com um objeto é realizada exclusivamente através de seus métodos, sem a necessidade de entender detalhes internos, o que favorece a colaboração em equipe e a construção de projetos complexos de maneira mais eficiente. Essa combinação de características faz da programação orientada a objetos uma escolha valiosa para o desenvolvimento de grandes projetos e o trabalho colaborativo entre desenvolvedores.

```python
# Exemplo de uso do paradigma
# orientado a objetos:

pessoa1 = Pessoa(nome="Edward Newgate")
pessoa2 = Pessoa(nome="Macaco Luffy")
cachorro1 = Cachorro(nome="Terremoto",
                        dono=pessoa1)
cachorro2 = Cachorro(nome="Buda de Ouro
                Gigante", dono=pessoa2)

cachorro2.morder(cachorro1)
pessoa1.xingar(pessoa2)
pessoa2.pegar_cachorro(cachorro1)
pessoa2.correr()
```

Programação Funcional

A programação funcional, um tipo de programação declarativa, difere das abordagens imperativas ao estruturar um programa através da declaração de funções, assemelhando-se a funções matemáticas. Essa abordagem possui o mais alto nível de abstração entre as três. Uma característica fundamental da programação funcional é o tratamento das funções como "cidadãs de primeira classe", o que significa que elas podem ser atribuídas a constantes e variáveis, passadas como parâmetro para outras funções e retornadas por funções, como qualquer outro tipo de dado.

Uma característica marcante de linguagens puramente funcionais é a ausência do conceito de atribuição, eliminando a troca de valores de variáveis. Essa escolha é feita devido à complexidade que as atribuições trazem ao código, uma vez que é preciso levar em consideração o momento de execução para saber o valor de uma variável. Por isso, linguagens puramente funcionais evitam o uso de *loops*, recorrendo à recursão e a funções padrão da linguagem como substitutos, uma vez que *loops* necessitam do conceito de atribuição.

Outro aspecto distintivo é a presença de funções puras, que são determinísticas e sempre retornam o mesmo valor para o mesmo conjunto de argumentos. Essa previsibilidade é possível devido à ausência de dependência de fatores externos além dos argumentos, o que torna as funções independentes entre si. Essa independência facilita a programação concorrente, onde funções podem ser executadas simultaneamente sem interferências.

A proximidade da programação funcional com a matemática confere-lhe uma vantagem na redução de *bugs* e na possibilidade de verificação formal. No entanto, essa proximidade também pode resultar em um nível de abstração que, em conjunto com o distanciamento do real funcionamento de um computador, torna os programas mais difíceis de entender.

Embora linguagens predominantemente funcionais não alcancem a mesma popularidade de outras, é notável que muitos dos conceitos inovadores desse paradigma, quando integrados a abordagens de diferentes paradigmas, revelam-se extremamente poderosos. Essas ideias, combinadas de maneira sinérgica, são amplamente adotadas por linguagens mais difundidas e tem bastante espaço no desenvolvimento moderno.

```haskell
-- Exemplo de uso do paradigma funcional
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)

dobrarNumero :: Int -> Int
dobrarNumero x = 2 * x

main = do
    let x = fatorial 3    -- x = 6
```

```
let y = [1, 2, 3, 4, 5]
let z = map dobrarNumero y
-- z = [2, 4, 6, 8, 10]
```

6.4 Exemplos Reais

C/C++

A linguagem C é notável por sua posição como uma linguagem de programação de nível mais baixo, sendo uma linguagem procedural amplamente utilizada. Ela desempenha um papel fundamental na construção de sistemas operacionais, *drivers*, sistemas embarcados e até mesmo na implementação de compiladores e interpretadores para outras linguagens, como *Python*. Grande parte do que é usado hoje no dia a dia foi construído nas bases sólidas dessa linguagem. Seu impacto não se limita apenas à sua aplicação direta, pois influenciou a criação de várias outras linguagens populares, como *JavaScript* e *Java*.

A simplicidade inerente à linguagem C é evidente em seu reduzido conjunto de palavras reservadas, tornando-a uma escolha simples para muitos desenvolvedores. No entanto, sua simplicidade também se reflete em características que podem desafiar desenvolvedores, como o gerenciamento manual de memória, utilizando ponteiros para referenciar posições de memória. Essa abordagem pode levar a códigos mais propensos a erros de segurança. Dentre os exemplos presentes nesta seção, C e C++ são as únicas linguagens que não usam coletor de lixo para a desalocação automática de memória inutilizada.

Outro aspecto peculiar é a falta de estruturas de dados mais complexas, e a inexistência de um tipo de dado dedicado para *strings*, que são representadas como simples vetores de caracteres. A tipagem estática e fraca da linguagem C significa que, embora o compilador faça verificações de tipo, a primitividade dos tipos permite conversões implícitas. Caracteres, *booleanos* e ponteiros são representados como inteiros, proporcionando facilidade de conversão entre eles.

C++ surge como um superconjunto de C, acrescentando suporte à Programação Orientada a Objetos (POO). Apesar de compartilhar muitas características com C, como sua aplicação, a curva de aprendizado íngreme e uma tipagem mais forte também a diferenciam de seu subconjunto. Enquanto C é venerada por sua velocidade, C++ expande suas capacidades, especialmente em domínios onde a POO é essencial.

```
// C:

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

int somar(int *vetor, int tamanho) {
    int soma = 0;
    for(int i=0; i < tamanho; i++)
        soma += vetor[i];
    return soma
}

int main() {
    int tamanho;
    printf("Digite o tamanho do vetor:");
    scanf("%d", &tamanho);

    int *vetor = malloc(tamanho * sizeof(int));
    if(vetor == NULL) {
        printf("A alocaçao de memoria falhou.\n");
        return 1;
    }

    printf("Digite %d inteiros:\n", tamanho);
    for(int i=0; i < tamanho; i++)
        scanf("%d", &vetor[i]);
    printf("A soma dos inteiros é: %d\n",
    somar(vetor, tamanho));

    free(vetor);
    return 0;
}

// C++:

#include <iostream>

int somar(int *vetor, int tamanho) {
    int soma = 0;
```

```
    for(int i = 0; i < tamanho; i++)
        soma += vetor[i];
    return soma;
}

int main() {
    int tamanho;
    std::cout << "Digite o tamanho do vetor: ";
    std::cin >> tamanho;

    int *vetor = new int[tamanho];
    if(vetor == nullptr) {
        std::cout << "A alocação de memória"
        + "falhou.\n";
        return 1;
    }

    std::cout << "Digite " << tamanho <<
    " inteiros:\n";
    for(int i = 0; i < tamanho; i++)
        std::cin >> vetor[i];
    std::cout << "A soma dos inteiros é: "
    << somar(vetor, tamanho) << std::endl;

    delete[] vetor;
    return 0;
}
```

JavaScript

JavaScript (JS) é uma linguagem de alto nível, reconhecida como uma das, se não a linguagem mais popular do mundo. Sua popularidade é impulsionada pelo fato de ser uma tecnologia fundamental na *World Wide Web*, com mais de 98% dos sites utilizando JS no *front-end*. Todos os navegadores mais utilizados possuem uma *engine* dedicada à execução de código JS.

Esta linguagem é conhecida por sua tipagem dinâmica e extremamente fraca, resultando em comportamentos por vezes bizarros e inconsistentes. Além disso, JS é multiparadigma, oferecendo suporte à orientação a objetos baseada em protótipo, uma alternativa às classes que dá dinamismo e flexibilidade ao código, e à programação funcional, incluindo, inclusive, o uso de funções de primeira classe. JS brilha especialmente na programação orientada a eventos, fazendo uso de sua capacidade para assincronicidade e manipulação do HTML através do DOM.

```
// Conversão implícita em JS:

const a = [] + []       // a = ""
const b = [] + {}       // b = "[object Object]"
const c = false + []    // c = "false"
const d = "123" + 1     // d = "1231"
const e = "123" - 1     // e = 122
const f = "123" - "abc" // f = NaN
//(NaN significa "not a number", cujo tipo é number)
```

Apesar de fazer parte da especificação da linguagem (*ECMAScript*), muitos recursos importantes são fornecidos pelo sistema de execução, como no caso do *browser*. O *loop* de eventos não bloqueante, por exemplo, permite que, mesmo usando uma única *thread*, o JS trate de outras tarefas enquanto espera a resposta de APIs, bancos de dados ou outros agentes assíncronos. Isso torna a linguagem extremamente poderosa para lidar aplicações com grande I/O (entrada e saída), como interfaces.

O ecossistema do *JavaScript* é vasto, incluindo *frameworks* como *Next.js*, *Angular* e *Vue.js*, que são amplamente utilizados no desenvolvimento *front-end*. Outro destaque é a popularização do formato JSON, derivado do JS, mas adotado como padrão de mercado por inúmeras linguagens.

JavaScript é considerada uma linguagem interpretada, embora a realidade seja mais complexa hoje em dia. No motor V8, presente nos navegadores baseados em *Chromium* (*Google Chrome*, *Microsoft Edge*, *Opera*, *Samsung Internet*), o código é convertido em *bytecode*, que é interpretado inicialmente. À medida que o programa é executado, partes frequentemente usadas são convertidas para código de máquina nativo, garantindo maior rapidez na execução. Isso é chamado de compilação *Just-In-Time* (JIT). Apesar disso, a execução de um programa JS ainda é feito a partir do código-fonte.

O motor V8, sendo independente de um navegador, possibilitou a expansão do uso do *JavaScript* para o *back-end*, através de ambientes como o *Node.js*, onde JS é usado com eficiência em aplica-

ções com grande quantidade de I/O. No *back-end*, JS também possui um amplo ecossistema, com o *npm* registrando o maior número de pacotes do mundo. *Frameworks* notáveis incluem *Express*, *Fastify* e *Nest.js*.

Assim como C, *JavaScript* também possui um superconjunto: *TypeScript*. *TypeScript* adiciona a segurança e a robustez da tipagem estática ao *JavaScript*, gerando código JS após a compilação. Essa evolução torna JS uma linguagem mais versátil, pois seu uso em sistemas de grande porte é favorecido.

```
// JS no Front-End:

document.getElementById('botaoMudarTexto')
.addEventListener('click', function() {
  const elementoDeOutput =
  document.getElementById('output')
  elementoDeOutput.innerText
  = 'Texto mudado!'
})


// JS no Back-End:

const http = require('http')

const server = http.createServer((req, res)
=> {
  res.writeHead(200, { 'Content-Type':
                       'text/plain' })
  res.end('Massa, meu irmão\n')
})

const port = 3000
server.listen(port, () => {
  console.log(`Servidor rodando em
  http://localhost:${port}/`)
})


// TypeScript:

function addNumbers(a: number, b: number):
number {
  return a + b;
}

const num1: number = 5;
const num2: number = 10;
const result: number =
addNumbers(num1, num2);
```

Python

Python é uma linguagem de alto nível, multiparadigma e extremamente versátil, destacando-se em diversas áreas de aplicação. Ao lado de *JavaScript*, tornou-se uma das linguagens mais populares no cenário da programação moderna. Uma característica marcante do *Python* é o seu foco na legibilidade e simplicidade do código. Para alcançar esse objetivo, a linguagem utiliza a identação como delimitador de bloco, em contraste com o uso de chaves em outras linguagens. Além disso, o uso de parênteses é minimizado, e muitos símbolos presentes em outras linguagens são substituídos por palavras do inglês, facilitando a compreensão do código.

```
Zen do Python:
- Bonito é melhor que feio.
- Explícito é melhor que implícito.
- Simples é melhor que complexo.
- Complexo é melhor que complicado.
- Linear é melhor do que aninhado.
- Esparso é melhor que denso.
- Legibilidade conta.
- Casos especiais não são especiais o
  bastante para quebrar as regras.
- Ainda que praticidade vença a pureza.
- Erros nunca devem passar silenciosamente.
- A menos que sejam explicitamente
  silenciados.
- Diante da ambiguidade, recuse a tentação
  de adivinhar.
- Dever haver um — e preferencialmente
  apenas um — modo óbvio para fazer algo.
- Embora esse modo possa não ser óbvio
  a princípio a menos que você seja holandês.
- Agora é melhor que nunca.
- Apesar de que nunca frequentemente é melhor
  do que *exatamente* agora
- Se a implementação é difícil de explicar,
  é uma má ideia
- Se a implementação é fácil de explicar,
  pode ser uma boa ideia
```

```
- Namespaces são uma grande ideia — vamos
  ter mais dessas!
```

Apesar de ser uma linguagem de tipagem dinâmica, o *Python* permite a utilização de anotações de tipo. Essas anotações servem não apenas para documentação, mas também para a verificação de tipos por meio de ferramentas externas, proporcionando um ambiente mais robusto para o desenvolvimento. Embora *Python* seja majoritariamente uma linguagem de tipagem forte, é importante notar que ela realiza conversões implícitas de outros tipos para *booleano*, demonstrando uma flexibilidade que pode ser explorada de maneira estratégica.

```
def lista_tem_elementos(lista: List) ->
bool:   # anotações de tipo
    if lista:    # lista é convertida
                 # para booleano
        return True
    else:
        return False
```

Python é considerada uma linguagem interpretada, porém, na implementação mais difundida, *CPython*, o código-fonte é compilado para *bytecode* e, após isso, interpretado. Apesar desse processo, a execução de um programa ocorre a partir do código-fonte, com ambas as etapas sendo realizadas unitariamente.

Python é amplamente reconhecido por sua riqueza em recursos e pela presença de uma vasta biblioteca padrão. Além disso, a instalação de pacotes externos é facilitada pelo gerenciador de pacotes *pip*, e esses pacotes são hospedados no registro *PyPI*, que conta com os pacotes feitos por uma das maiores comunidades na esfera do desenvolvimento.

A linguagem é amplamente utilizada no desenvolvimento *web back end*, sendo suportada por *frameworks* como *Django* e *Flask*. Além disso, *Python* desempenha um papel crucial em campos como inteligência artificial e aprendizado de máquina, contando com bibliotecas renomadas como *TensorFlow*, *Keras* e *PyTorch*. Na área de ciência de dados, *Python* é uma escolha popular, impulsionada por bibliotecas como *NumPy*, *Pandas* e *Matplotlib*, que oferecem ferramentas poderosas para análise e visualização de dados, além das bibliotecas de IA citadas. Adicionalmente, *Python* é frequentemente empregado na automação, onde uma variedade de bibliotecas pode ser explorada para simplificar e otimizar tarefas em diferentes contextos.

```
# Python:

def soma_quadrados_se_par(lista):
    return sum(x**2 for x in lista if x % 2 == 0)

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
resultado = soma_quadrados_se_par(numeros)
print(f"A soma dos quadrados dos números
pares em {numeros} é: {resultado}")
```

Java

Java é uma linguagem de programação multiparadigma, mas se destaca por sua ênfase em orientação a objetos. Em sua estrutura, todo código em um programa Java precisa estar encapsulado em classes, sendo cada arquivo responsável por definir uma classe específica. Com sua principal aplicação voltada para a construção de *software* robusto em larga escala no *back-end*, Java tem muito espaço no mercado nesse contexto.

Sua arquitetura é híbrida entre interpretação e compilação, sendo que o código Java é compilado para *bytecode*, que, por sua vez, é interpretado pela Máquina Virtual do Java (JVM), um aspecto fundamental da linguagem. Diferentemente de linguagens como *JavaScript* e *Python*, onde o código-fonte é executado diretamente, em Java é o arquivo em *bytecode* (.class) que é processado pelo usuário. Isso significa que, ao contrário de outras linguagens, o *bytecode* Java pode ser manipulado pelo usuário.

O lema "escreva uma vez, execute em qualquer lugar"(WORA) destaca a portabilidade do *bytecode* Java. Uma vez compilado, o *bytecode* pode ser executado em qualquer ambiente que possua a JVM, demonstrando a versatilidade da linguagem. É interessante observar que Java, cuja compilação para *bytecode* esteve presente desde seu início, foi responsável pela popularização desse método. Isso foi adotada por linguagens como *JavaScript* e *Python* apenas posteriormente, por motivos de performance.

Java, ao possuir características tanto de linguagens interpretadas quanto compiladas, combina a portabilidade de uma linguagem interpretada com a privacidade e a tipagem estática típica de linguagens compiladas. Contudo, apesar de suas vantagens, Java também enfrenta críticas. Sua sintaxe verbosa e menos amigável para iniciantes é frequentemente apontada como uma barreira à entrada na linguagem. Além disso, alguns problemas de desempenho relacionados à JVM são citados como desafios a serem superados.

```java
public class Main {
    public static void main(String[] args) {
        Estudante hugo = new
        Estudante(123456, "Caike Vinicius
        dos Santos");
        Estudante mestreDeObras = new
        Estudante (654321, "Vinicius
        Silva Castro");

        Curso cursoDeJava = new Curso
        ("Aprenda a Fazer o Minecraft 2");
        cursoDeJava.matricular(hugo);
        cursoDeJava.matricular(mestreDeObras);

        List<Estudante> alunos =
        cursoDeJava.getAlunos()
        System.out.println("Alunos: "
        + alunos);
    }
}
```

6.5 Bibliografia

[1] Compiled vs interpreted programming languages | what's the difference? - youtube. URL: https://youtu.be/F64_bwahaWQ?si=231_k91GpdvCPmJk.

[2] Static vs dynamic typing | which languages are better? - youtube. URL: https://youtu.be/GqXpFycPWLE?si=vXei9EWiijcXiTaQ.

[3] Static vs dynamic programming languages | what's the difference? - youtube. URL: https://youtu.be/bCIFTWQorL0?si=eHUNDQtGOmKLAial.

[4] Typing: Static vs dynamic, weak vs. strong / intro to javascript es6 programming, lesson 16 - youtube. URL: https://youtu.be/C5fr0LZLMAs?si=wLLbSbyw4SrbX5ia.

[5] Programming paradigm - wikipedia. URL: https://en.wikipedia.org/wiki/Programming_paradigm.

[6] Introduction of programming paradigms - geeksforgeeks. URL: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/.

[7] Dear functional bros - youtube. URL: https://youtu.be/nuML9SmdbJ4?si=DT0aT1OfnTfsfvaJ.

[8] Functional programming - a general introduction - youtube. URL: https://youtu.be/8z_bUIl_uPo?si=ER_PFJvuvp8Gdv2Y.

[9] C (programming language) - wikipedia. URL: https://en.wikipedia.org/wiki/C_(programming_language).

[10] C++ - wikipedia. URL: https://en.wikipedia.org/wiki/C%2B%2B.

[11] Javascript - wikipedia. URL: https://en.wikipedia.org/wiki/JavaScript.

[12] Usage statistics of javascript as client-side programming language on websites, february 2024. URL: https://w3techs.com/technologies/details/cp-javascript.

[13] Understanding the v8 javascript engine - youtube. URL: https://youtu.be/xckH5s3UuX4?si=ox4aZI4zr6tA2tJM.

[14] Franziska hinkelmann: Javascript engines - how do they even? | jsconf eu - youtube. URL: https://youtu.be/p-iiEDtpy6I?si=FqMWP5J8KeWzs4Xx.

[15] Typescript - wikipedia. URL: https://en.wikipedia.org/wiki/TypeScript.

[16] Python (programming language) - wikipedia. URL: https://en.wikipedia.org/wiki/Python_(programming_language).

[17] Python data science handbook | python data science handbook. URL: https://jakevdp.github.io/PythonDataScienceHandbook/.

[18] Zen de python – wikipédia, a enciclopédia livre. URL: https://pt.wikipedia.org/wiki/Zen_de_Python.

[19] Python in 100 seconds - youtube. URL: https://youtu.be/x7X9w_GIm1s?si=GlIHBCQoMrM9nR66.

[20] Java (programming language) - wikipedia. URL: https://en.wikipedia.org/wiki/Java_(programming_language).

[21] Java in 100 seconds - youtube. URL: https://youtu.be/l9AzO1FMgM8?si=ogxWVD0mySKl6uiH.

Parte II

Projetos

Capítulo 7

upGrade

Mauricio Cândido de Souza

7.1 Conceito do projeto

Com o intuito de criar uma ferramenta que é utilizada pelos estudantes de faculdades ou cursos com matérias separadas em diversos períodos contendo dependências entre elas, esse projeto se propõe a apresentar uma interface gráfica, completamente dinâmica e customizável para que alunos dos mais diversos cursos possam organizar e planejar seu trajeto futuro independente de sua instituição de ensino.

7.2 Pré-requisitos e recursos utilizados

Esse projeto depende fortemente do *Cytoscape*, utilizado para visualização e organização dos dados no projeto. Não é necessária nenhuma instalação do mesmo, já que ele é utilizado via um script hospedado na cloudflare.

7.3 Passo a passo

O projeto foi executado em algumas etapas breves:

1. Criação da estrutura básica web e estrutural, criando os arquivos **index.html** e **index.css**, além disso também definindo o padrão do arquivo **grades.jsonc**.
2. Criação da lógica inicial transferindo o objeto para um conjunto de _*nodes*_ dentro do _*cytoscape*_, contida no **index.js**
3. Desenvolvimento do algoritmo de caminho das dependências, buscando minimizar conflitos e melhorar visibilidade numa forma genérica.
4. Desenvolvimento da funcionalidade de visualização da "árvore de dependências"e alteração de perfis de matérias.

7.4 Instalação / Execução

O projeto não necessita de nenhuma instalação prévia propriamente dita, as funcionalidades do projeto (como a leitura de arquivos JSON) e conexão requerem que os arquivos estejam hospedados em um "servidor".

Portanto, caso queira subir o projeto localmente, clone ou baixe o repositório e, estando dentro da pasta, inicie esse ambiente como um servidor. Para fazer isso recomendo a utilização da extensão Live Server do Visual Studio Code, mas qualquer outra alternativa de subir um servidor localmente para lidar com o CORS funciona.

Dentro do código, é carregado sempre a grade existente no arquivo **grades.jsonc**, caso queira alterar, basta alterar essa configuração no código e apontar para seu arquivo, ou substituir o arquivo com o seu próprio. O formato de uma grade deve ser respeitada para ser processada corretamente.

7.5 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

```
https://github.com/mauriciocsz/upGrade
```

7.6 Imagens/screenshots

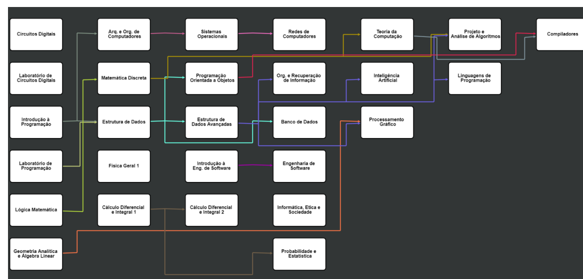

Figura 7.1: Figura 1: Interface base do programa, com a grade atual de Ciência da Computação da UFSCar - Sorocaba

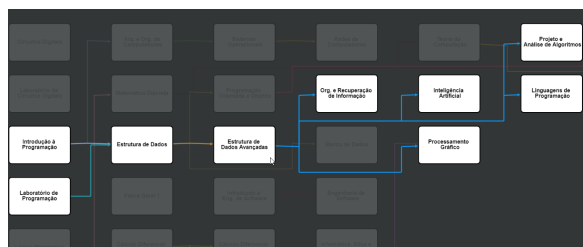

Figura 7.2: Figura 2: Exibição da trilha de dependências entre as disciplinas da grade

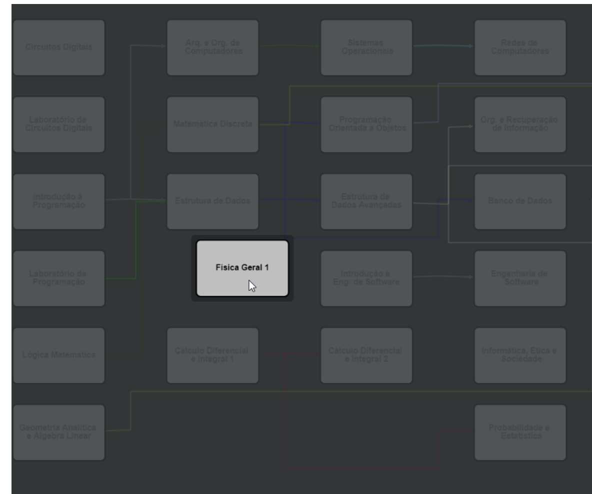

Figura 7.3: Figura 3: Exibição da *feature* de clicar e arrastar as disciplinas da grade

Capítulo 8

P3 APP: Aplicação web para gerenciamento de tempo e tarefas

Pedro Gonçalves Correia
Pedro Enrico Nogueira Barchi
Pedro Henrique Alves de Araujo Silva

8.1 Conceito do projeto

Este projeto foi realizado com a finalidade de projetar uma aplicação capaz de agregar em seu interior uma única ferramenta para que seus usuários possam criar e gerenciar anotações e despesas, e cronometrar períodos de estudo conforme a metodologia *pomodoro*. Com o intuito de estudar e se aprofundar em conceitos do desenvolvimento *web* e como ele pode se relacionar com a cibersegurança, para isto, nossa aplicação conta com um sucinto sistema de *login* que utiliza conceitos de segurança para garantir que cada usuário possa ter acesso único aos seus arquivos.

8.2 Pré-Requisitos e recursos utilizados

O grupo utilizou as linguagens *JavaScript*, HTML, CSS e *React* para o desenvolvimento do projeto, utilizando as *bibiliotecas* que estão descritas em **package-lock.json**

8.3 Passo a passo

1. Criação e desenvolvimento de protótipo utilizando a ferramenta *Figma*
2. Criação da página de ínicio
3. Criação da página Notas
4. Criação da página Finanças
5. Criação da página *Timer*
6. Criação do *login*, do cadastro e da autenticação

8.4 Execução

Para executar o projeto deve ser instalado localmente o **node.js**, e com isso ao abrir os arquivos tanto do *front* quanto do *back* devem ser instaladas todas as as dependências pelo comando: **npm install**

Para que o *app* funcione em sua totalidade é necessário também fazer a importação do *dump* do banco de dados localizado em P3-BD em algum *workspace mysql*, para isso é necessário que o *mysql* também esteja instalado.

Obs: nessa etapa é importante configurar os *models* do *backend* de acordo com suas configurações de banco de dados local, para isso entre nos arquivos *models* de **P3-back/src/models** e em cada arquivo ajustar de acordo com usuário e senha local do *mysql* no *Sequelize*.

Para iniciar o projeto deve ser disparado o seguinte comando tanto no *front* quanto no *back*: **npm start**

8.5 Imagens/screenshots

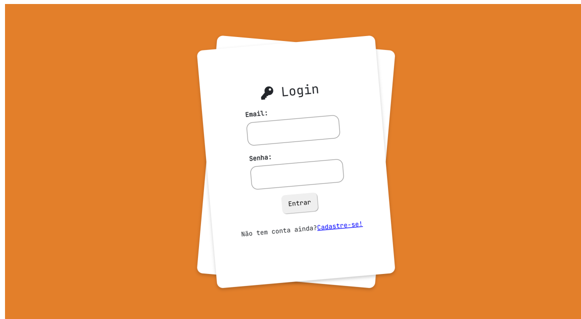

Figura 8.1: Figura 1: Página de *login* e autenticação do programa

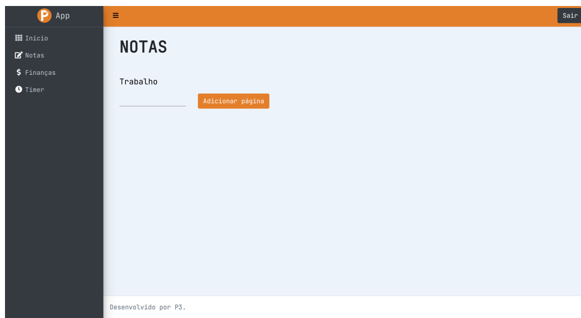

Figura 8.2: Figura 2: *Landing page*. Por padrão leva à parte de Notas.

Figura 8.3: Figura 3: Notas preenchidas e armazenadas no sistema

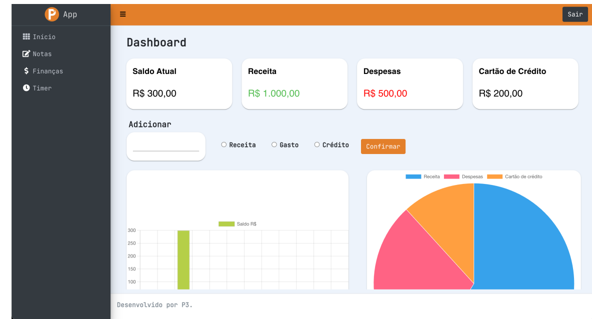

Figura 8.4: Figura 4: *Dashboard* da carteira de finanças e gastos

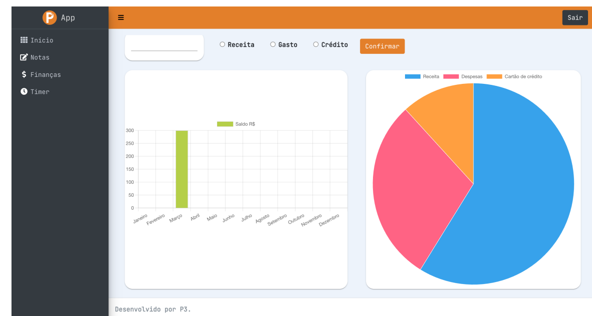

Figura 8.5: Figura 5: Geração de gráficos com base nos dados inseridos na carteira de finanças e gastos

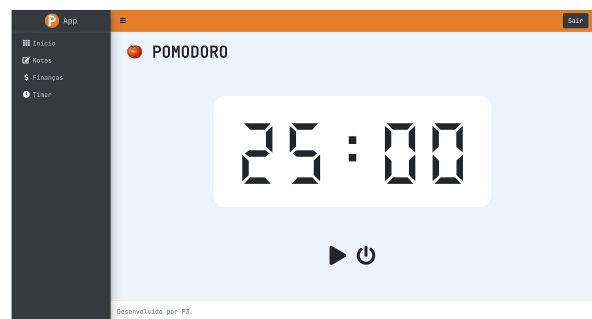

Figura 8.6: Figura 6: *Timer* configurado com a metodologia *pomodoro*. É possível ajustar o tempo do *timer*.

8.6 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

https://github.com/Pedro-bnogueira/P3-app

Capítulo 9

Dashboard Carteira de Investimento

Alex Sandro Momi Junior

9.1 Conceito do projeto

Esta é uma aplicação para o monitoramento de carteira de investimentos, que possuam ativos em diversas corretoras, em um único. Ela fornece uma visão geral da distribuição do portfólio, da receita recebida e do crescimento do patrimônio.

9.2 Pré-Requisitos e recursos utilizados

Para o desenvolvimento do *dashboard* foi utilizado *Python* e *MySQL* como tecnologias principais. Além de utilizar as seguintes bibliotecas:

- ***seaborn*** e ***plotly***: para criação dos gráficos;
- ***yfinance***: para buscar as informações sobre os ativos;
- ***pandas***: para organizar as informações recuperadas pelo *script* de *webscrap*;
- ***streamlit***: para criação do *Data App*.

9.3 Instalação e Execução

Para conseguir executar o projeto é preciso ter o *Docker* e *docker-compose* V2 instalados. Após instalar ambos é só executar os comandos abaixo:

```
# Constrói as imagens
docker compose build

# Inicia os serviços
docker compose up
```

Após esses comandos o serviço será iniciado, utilizando a porta 8051 para a aplicação e 3306 para o banco de dados *MySQL*.

OBS: Os comandos acima estão considerando a utilização do *docker compose* V2. Caso você tenha instalado a primeira versão é só utilizar os comandos com *"**docker-compose ...**"* ao invés de *"**docker compose ...**"* .

9.4 Funcionalidades

Distribuição Atual da Carteira

A primeira funcionalidade do *Dashboard* é mostrar, de maneira simples e objetiva, a distribuição atual do portfólio de investimentos. A distribuição é exibida por duas óticas distintas:

- **Por Ativo**: Apresenta a participação de cada ativo na sua carteira, oferecendo uma visão de onde estão concentradas suas maiores posições.

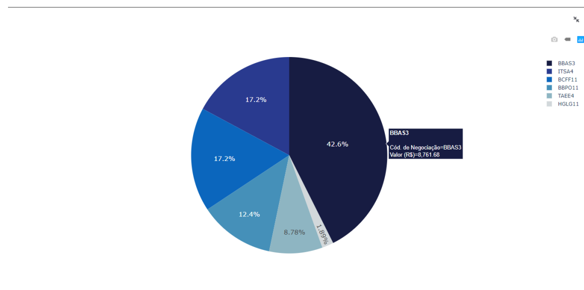

Figura 9.1: Figura exibindo a visão "Por Ativo" da carteira. Clique aqui para ver a imagem ampliada.

- **Por Classe de Ativo**: Exibe a distribuição do portfólio por classes de ativo, auxiliando na melhor análise da sua estratégia de investimento.

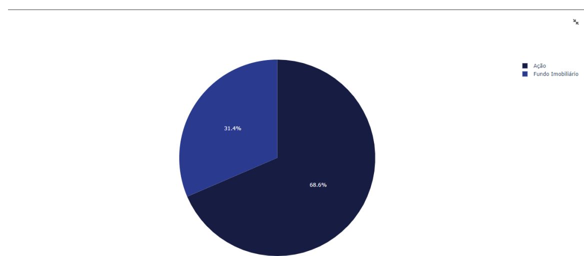

Figura 9.2: Figura exibindo a visão "Por Classe de Ativo" da carteira. Clique aqui para ver a imagem ampliada.

Rendimentos

Os rendimentos são um aspecto vital do investimento, especialmente para investidores de longo prazo que buscam um fluxo constante de renda passiva. Para facilitar um monitoramento eficaz desses rendimentos, o *Dashboard* apresenta dois gráficos distintos.

O primeiro gráfico apresenta os rendimentos mensais, compilando a renda recebida a cada mês e fornecendo informações detalhadas sobre cada desembolso, incluindo a data e a natureza da receita. O segundo gráfico agrega os rendimentos de cada ativo, ilustrando os ganhos totais recebidos desde o momento da compra.

Rendimentos Mensais

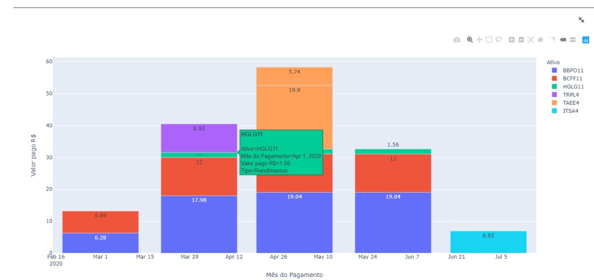

Figura 9.3: Figura exibindo a visão "Rendimentos Mensais" da carteira. Clique aqui para ver a imagem ampliada.

Rendimentos por Ativo

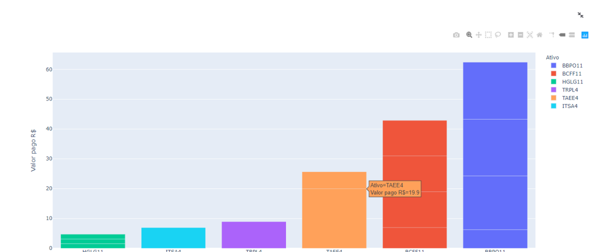

Figura 9.4: Figura exibindo a visão "Rendimentos Por Ativo" da carteira. Clique aqui para ver a imagem ampliada.

9.5 Crescimento do Patrimônio

Por fim, mas não menos importante, o gráfico de acumulação de patrimônio oferece uma perspectiva concisa sobre seus **investimentos** e seu **valor bruto** conforme eles evoluem ao longo do tempo.

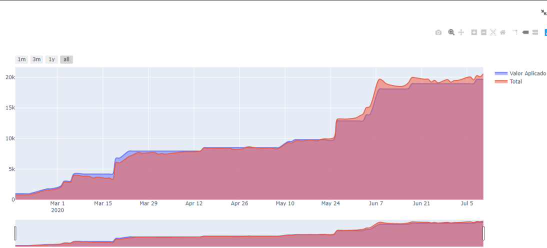

Figura 9.5: Figura exibindo a visão "Crescimento Patrimonial" da carteira. Clique aqui para ver a imagem ampliada.

9.6 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

`https://github.com/AlexJunior01/dashboard_carteira_de_acoes/`

Capítulo 10

BCC-HUB : Plataforma de Hospedagem Web Autogerenciada da Computação

Rafael Gimenez Barbeta

10.1 Conceito do projeto

Durante o 1º Semestre de 2022, os calouros da computação tinham à sua disposição um site de dúvidas para a disciplina de Introdução a Programação. O site era hospedado "de graça" na plataforma conhecida como *Heroku*. Infelizmente, para a monitoria de 2023 essa plataforma já não estava mais gratuita. Foi feita uma busca por outras opções gratuitas para hospedagem do site, sem sucesso. Todas as opções "*free*" impõem algum tipo de restrição quanto ao uso, como limite de tempo, de banda larga etc. Além do site da monitoria, muitos alunos também desenvolvem seus próprios projetos *web*, tanto para o HackoonSpace, Maritacas GameDev, entre outras entidades do curso, ou mesmo para seu próprio portfólio. Dessa questão surgiu a ideia do projeto : Criar um servidor próprio do curso para comportar as criações dos alunos, grátis, autogerenciado e centralizado, o **BCC-HUB**!

De forma simplória, um "servidor" nada mais é do que uma máquina executando 24 horas e publicamente acessível na *internet*. Apesar de na teoria parecer simples, gerenciar e configurar um servidor físico pode ser bastante complexo, envolvendo não só configurações locais como configurações a nível de rede. Foi necessário o intermédio dos professores e técnicos de TI da UFSCar para tornar esse projeto possível, além das definições explicitadas nesse *README*. O computador que atua como servidor também foi emprestado.

O projeto, portanto, tem como foco preparar e configurar todas as dependências da máquina para receber novos projetos e disponibilizá-los... de graça!

O restante desse *README* explicará, de forma simplificada, como essas configurações foram feitas, e sempre que possível, será mostrado as configurações do servidor real no ar. Além disso, deixei um "guia"para quem quiser montar seu próprio servidor web.

10.2 Pré-requisitos e recursos utilizados

Hardware

Qualquer computador de mesa no geral pode funcionar como servidor. Obviamente exitem *hardwares* feitos com esse propósito (e mais caros), mas no geral basta que tenha um processador decente, memória, placa de rede e possa executar sem parar.

Especificamente para esse trabalho, foi utilizado o seguinte computador: Microcomputador; tipo servidor, c/ processador *Xeon X3430 Quad Core*, RAM 8 GB, HD 1,25 TB. - Marca IBM

Recursos de Rede

Para ser "encontrado"na *internet* você precisará de um IP público. Computadores em redes domésticas obtém um tipo de IP que é chamado de privado pelo roteador e que não são roteáveis na *internet*. Isso porque a quantidade de endereços IPv4 estão esgotados hoje em dia.

No caso desse servidor, o IP obtido pelos computadores da universidade já é um endereço público e protegido por *firewall*.

Talvez seja necessário realizar um port-forwarding se desejar executar em um computador de casa, ou utilizar um *proxy* reverso como o ngrok.

Software

Uma diversidade grande de *softwares* são executados em conjunto para garantir o funcionamento do servidor. Abaixo, deixo um breve resumo de cada um deles:

- *nginx* : servidor-web, com capacidade para atuar como *gateway* de aplicação e balanceador de carga. Ele é utilizado por "redirecionar"um pedido a um site específico hospedado no *hub* para um *container docker*, em que a aplicação/site está "hospedado". Ele é a ponte entre os sites e a *internet*. Além disso, nele foi configurado certificado SSL e o WAF para requisições inseguras "descriptografadas", ou maliciosas. Esse *README* falará mais sobre esses componentes abaixo.
- *sshd* : servidor de SSH para acesso remoto. O SSH permite que os administradores e usuários possam acessar o *hub* remotamente, executar comandos e gerenciar configurações. A porta do serviço em questão, a 22, só está liberada por meio da VPN da universidade, e a autenticação é feita por meio de chaves públicas ED25519
- *ufw*: *firewall* de *host*, impede conexões indevidas em portas específicas. Consegue *filtrar* IPs e até mesmo cabeçalho TCP. Foi configurado para negar acessos que não sejam por meio dos endereços privados da VPN da universidade, no caso do *firewall* principal falhar.
- *fail2ban* : ferramenta de banimento automático com base em *logs* de erros de aplicações. É utilizado em conjunto com o WAF para banir IPs que tenham excedido um limitar definido de requisições maliciosas "permitidas". Ele simplesmente lê o *log* de auditoria gerado pelo WAF e impede acesso se muitas requisições marcadas como maliciosas foram enviada
- *daemon docker*: serviço de execução de *containers docker*. Um *container* é semelhante a uma máquina virtual, em termos que ele permite isolar processos, que são programas em execução, do resto do sistema. É mais rápido que uma VM, uma vez que o *kernel* é compartilhado do "hospedeiro". A vantagem de executar as aplicações e sites e *containers* é que eles são auto contidos, e já possuem os *softwares* que são necessários para executar as aplicações. Além disso, as dependências de um projeto não interferem com a de outro, pois são ambientes isolados, o que permite melhor escalabilidade e gerenciabilidade do ambiente.
- *modsecurity*: é o WAF mencionado acima, ele é um módulo criado originalmente para o *apache* que foi "conectado"ao *nginx*. Esse módulo inspeciona cada requisição HTTP procurando indícios de atividade maliciosa, e se detectado , ele impede que a requisição sequer chegue em um dos *containers*. As regras de detecção foram providenciadas pela OWASP através da coreruleset.

Home-Page

Foi desenvolvida uma *home-page* para o projeto também, e ela se encontra nesse repositório do *Github*. Ela é uma adaptação do *template* providenciado por html design.

10.3 Resumo de configurações

Abaixo, deixarei um resumo de passos feitos para configurar o *hub*, incluindo a instalação de componentes e sua execução. As etapas estão aproxima-

damente em ordem, mas algumas configurações foram feitas em ordem distinta do que aqui apresentado. Porém, para melhor compreensão, elas foram separadas em blocos.

Instalação do SO e configuração da BIOS

Etapa realizada no laboratório, com acesso físico ao servidor. Instalação rotineira de sistema operacional. Foi utilizado a distribuição Debian, mais precisamente a versão 11 "*bullseye*" . Além da própria instalação, a BIOS foi configurada para que o computador religue sozinho em caso de falta de energia, garantindo a disponibilidade do servidor.

O primeiro usuário criado na máquina foi o "monitor", com nome "*bcchub*" .

Conectividade, alcançabilidade e acesso remoto

Esta parte foi feita com ajuda do professor Fábio Luciano Verdi e do analista de TI da UFSCar. As etapas estão descritas abaixo. Note que algumas delas ocorreram ao longo de quase todo o projeto.

1. Atrelagem do MAC da interface *ethernet* do servidor a um IP fixo, distribuído pelo DHCP. Aqui, fixamos o IP para que ele nunca mude. O computador sempre obterá o mesmo IP do DHCP

2. Atualização do *firewall*, permitindo tráfego de rede nas portas 80 e 443 do servidor. Apesar do IP obtido ser público (ou seja alcançável na *internet*), qualquer acesso externo é barrado pelo *firewall*. As regras precisaram ser atualizadas para que os usuários possam acessar as páginas *web*.

3. Criação de "conta"na VPN da universidade, e liberação da porta 22. O acesso remoto é feito por *ssh*, na porta 22. Porém, essa porta só não é bloqueada pelo *firewall* se a conexão partir da conexão por VPN da universidade, por segurança.

4. Obtenção de nome de domínio, por cima do subdomínio *dcomp*. Uma entrada no DNS da universidade foi inserido para ser possível acessar o *hub* por nome, ao invés de IP, etapa essa necessária para futuramente obter um certificado SSL. O *hub* é acessível pelo nome bcchub.dcomp.ufscar.br. O nome foi decidido a partir de um formulário enviado a todos os estudantes ativos na graduação em Ciência da Computação.

Como mencionado, um serviço *ssh* foi instalado para permitir conectividade:

```
sudo apt install sshd
```

Em seguida, desabilitou-se o acesso por senha, por segurança. Para isso, editou-se **"sshd_config"** e o atributo ***PasswordAuthentication*** foi setado como **"no"** . Além disso, no **"*home*"** do usuário monitor, adicionou-se a chave pública do administrador:

```
echo "<chave_publica_aqui>"
>> ~/.ssh/authorized_keys
```

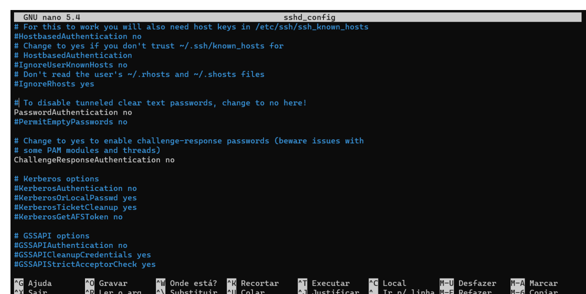

Figura 10.1: Figura exibindo a alteração da configuração ***PasswordAuthentication*** para **"no"** . Clique aqui para ver a imagem ampliada.

A partir desse ponto, todas as configurações foram feitas remotamente, dentro da VPN da universidade.

Se desejar replicar o projeto, será necessário realizar mais ou menos os mesmos passos. Cabe destacar que talvez seja necessário comprar um nome de domínio e IP.

Instalação do servidor web

Utilizou-se a versão *mainline* do *nginx*, versão 1.25. O tutorial de instalação se encontra aqui:

nginx.org.

Após a instalação, foi feita a configuração da *home page* do *hub* no *nginx*, cujo código fonte se encontra na íntegra aqui nesse repositório. Para isso, é necessário definir um arquivo contendo um **"server-block"** na pasta ***/etc/nginx/sites-available*** e criar um link simbólico para esse arquivo na pasta ***/etc/nginx/sites-enabled***. O arquivo principal, ***/etc/nginx/nginx.conf***, por ora, não precisa ser modificado, pois ele já inclui qualquer configuração de sites contidos na pasta *sites-enabled*. Abaixo, deixo um exemplo da configuração inicial para servir uma página web por HTTP.

```
server {
        listen 80 default_server;
        root /var/www/html;
        index index.html index.htm;
        server_name bcchub.dcomp.ufscar.br;

        location / {
            try_files $uri $uri/ =404;
        }

        # Outras configs...
        #...
        #...
}
```

Figura 10.2: Configuração inicial do ***BCC-HUB***. Clique aqui para ver a imagem ampliada.

Perceba que essa não é a configuração em execução no momento, que é "apenas" por HTTPS, explicado no tópico abaixo.

É boa prática testar se a configuração está sintaticamente correta antes de recarregá-la:

```
sudo nginx -t
```

Depois, recarregue para que as mudanças entrem em ação:

```
sudo systemctl reload nginx
```

Protegendo o servidor

O *hub* irá abrigar diversas aplicações de desenvolvedores distintos e essas estarão publicamente disponíveis na *internet*. Não se sabe de antemão que tipo de aplicação será hospedada, ou quais dados serão guardados. Por isso, se faz necessário criar uma estrutura segura de hospedagem, que possa atender a diferentes requerimentos de segurança das aplicações, além do próprio servidor. Se tratando de segurança *web*, há três problemas de segurança que merecem atenção:

1. Captura de pacotes
2. Vulnerabilidade aplicação
3. Vulnerabilidade do servidor

O item 1 faz referência à "bisbilhotagem"de pacotes entre o servidor e o cliente. Um ator de ameaça em algum ponto da comunicação pode ler os pacotes e extrair informações comprometedoras, como usuário e senha. Para evitar esse tipo de problema, se usa conexão criptografada, HTTPS.

Para poder servir páginas HTTPS precisamos de uma chave privada e um certificado digital para o domínio **bcchub.dcomp.ufscar.br**. O domínio foi validado com a autoridade de certificação ZeroSSL de graça. Outra boa opção gratuita é a Let's Encrypt, recomendada se quiser montar seu próprio servidor.

Com a chave e certificado em mãos, adicione eles na configuração do *nginx*. Um novo arquivo com um novo **"server_block"** deve ser adicionado em ***sites-available***:

```
server {
        # SSL configuration
        #
        listen 443 ssl default_server;
        ssl_certificate
        \<CAMINHO_PARA_O_CERTIFICADO\>;

        ssl_certificate_key
        \<CAMINHO_PARA_A_CHAVE\>;

        root /var/www/html;
        # Add index.php to the list
        # if you are using PHP
        index index.html index.htm
        index.nginx-debian.html;
        server_name bcchub.dcomp.ufscar.br;

        location / {
            try_files $uri $uri/ =404;
        }
}
```

Modificou-se o antigo *server block* HTTP para sempre redirecionar o usuário para a versão segura do site:

```
server {
        listen 80 default_server;
        server_name _;

        return 301
        https://bcchub.dcomp.ufscar.br;

}
```

Para mitigação do item 2, a opção escolhida foi instalar um *Web Application Firewall*, o *modsecurity*, mencionado anteriormente. Esse passo é um tanto trabalhoso, visto que é necessário compilá-lo como um módulo dinâmico para o *nginx*. Esse artigo foi utilizado como guia para realizar essa instalação.

Adicionou-se a diretiva ***load_module /etc/nginx/modules/ngx_http_modsecurity_module.so;*** para carregar o módulo e ativá-lo com ***modsecurity on***. Como mencionado, opera com a Core Rule Set(CRS) da OWASP, que para o projeto está instalada na pasta ***/usr/local/modsecurity-crs***. O repositório das regras se encontra aqui: https://github.com/coreruleset/coreruleset.git.

Na configuração principal do *modsecurity*, é possível definir se o módulo apenas fará detecção, registrando cada tentativa suspeita, ou ele efetivamente bloqueará a requisição. Esse comportamento é controlado pela diretiva ***SecRuleEngine*** e para esse projeto, foi definida como ***On*** (bloqueio ativo):

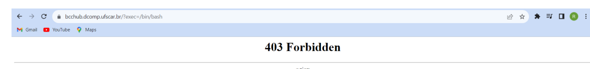

Figura 10.3: Bloqueio para tentativas suspeitas funcionando. Nesse caso, tentativa de acessar um *shell*. Clique aqui para ver a imagem ampliada.

Apesar da *corerulest* da OWASP cobrir diversos cenários de ataque, ela não impede que uma vulnerabilidade em uma aplicação hospedada seja explorada, apesar de dificultar o trabalho do ator ameaça. Pensando nisso, complementou-se a segurança das aplicações com o software *fail2ban*, que é capaz de ler *logs* de erro de programas e tomar uma ação de banimento com base em uma série de tentativas de ataque. A ideia é que um agente ameaça provavelmente fará uma série de solicitações maliciosas mal-sucedidas (identificadas pelo WAF) para tentar encontrar alguma vulnerabilidade e/ou "*bypassar*" o WAF. Isso desencadeará uma resposta do *fail2ban* que bloqueará qualquer outra requisição desse IP por um longo período de tempo, repelindo a ameaça por ora. Isso também é efetivo contra *scanners* de vulnerabilidade percorrendo toda a *internet* e procurando algum alvo vulnerável.

A instalação é feita diretamente pelo gerenciador de pacotes do *Debian*:

```
sudo apt install fai2ban
```

Adiciona-se uma "jaula" ao *fail2ban*, especificando o *log* monitorado, além de um "filtro", que é a expressão regular que se busca no *log* para identificar solicitação maliciosa mal-sucedida e repetida.

Arquivo ***jail.local***:

```
[nginx_modsec]
enabled  = true
filter   = nginx_modsec
action   = iptables-multiport[name=ModSec,
                              port="http,https"]
bantime  = <CENSURADO>
maxretry = <CENSURADO>
ignoreip = <CENSURADO>
logpath = /var/log/nginx/error.log
        /var/log/nginx/error.log.1
```

Arquivo ***filter.local***:

```
[Definition]
failregex = \[client <HOST>\] ModSecurity
```

HOST é o IP do solicitante. Se a expressão der "*match*"mais do que "x" vezes, ele será banido pelo tempo definido na "jaula".

A *screenshot* abaixo mostra quantos IPs já foram banidos. Todos eles são *scanners* de vulnerabilidade, percorrendo a web:

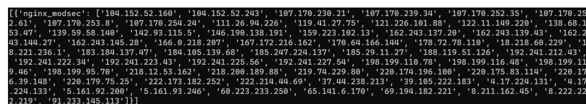

Figura 10.4: IPs banidos do site, identificados pelo *fail2ban*. Clique aqui para ver a imagem ampliada.

Por fim, temos o item 3. Se refere a problemas na própria configuração do *nginx* ou vulnerabilidades nesse *software*. A versão do *nginx* é relativamente recente e não aparenta ter nenhum *bug* conhecido. Porém, é necessário sempre atualizá-lo para evitar problemas.

Execução dos projetos em contêineres

Para poder "*hostear*" vários sites com diferentes *back-ends* e dependências, as aplicações são conteinerizadas em *dockers*. A instalação do *daemon* pode ser feita seguindo o passo a passo descrito na documentação oficial.

Além disso, para aumentar o nível de segurança e isolamento, os contêineres são executados a nível de usuário (*rootless mode*). Na prática, isso significa que cada usuário no *hub* só poderá interagir com seus próprios contêineres, e uma eventual invasão bem sucedida só terá acesso aos conteúdos dentro do *docker*, sem permissão de administrador "*root*" . Esse é mais um dos componentes de segurança do *hub*. A documentação oficial também explica o passo a passo para realizar esse procedimento.

No geral, define-se para as aplicações um ***Dockerfile*** e um ***docker-compose.yml***. Esses arquivos instruem o *docker* como iniciar o contêiner e executar a aplicação, além de fazer as conexões necessárias entre as portas "virtuais" do contêiner e as portas "reais" do *host*. Para "subir" a aplicação depois, execute:

```
docker compose up
```

Para cada aplicação/site no ar, é necessário definir nos arquivos de configuração do *nginx* como acessá-la. Essencialmente, cada aplicação pode ser acessada a partir de uma "rota" distinta no servidor. A título de exemplo, enquanto a *url* **https://bcchub.dcomp.ufscar.br** leva à *home page* do servidor, **https://bcchub.dcomp.ufscar.br/siteexemplo** irá ser roteada pelo *nginx* ao contêiner contendo "*siteexemplo*" . Esse contêiner irá processar a requisição e devolver para o *nginx*, que repassará para o cliente. Isso é possível pois o servidor, pelo caminho especificado, consegue distinguir uma requisição pela *home page* de outra para o "*siteexemplo*" . A figura abaixo ajuda a ilustrar esse conceito:

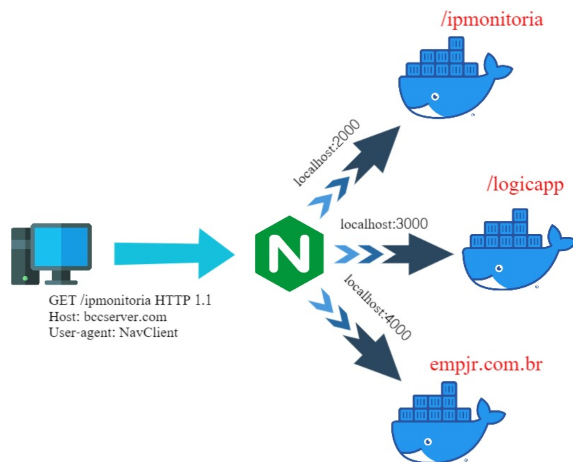

Figura 10.5: Multiplexação dos contêineres, implementada na infraestrutura do projeto. Clique aqui para ver a imagem ampliada.

Essa "multiplexação" se manifesta na forma da diretiva *proxy_pass* do *nginx*, sendo que o "destino" é o próprio *localhost*, em uma porta na qual o *docker* está "escutando" . Além disso, caso os estudantes do curso tenham seu próprio domínio e queiram usar invés do subdomínio *bcchub*, isso é possível através de um *"virtual host"* , basta que a entrada DNS aponte para o servidor e o *nginx* consegue encaminhar a solicitação para o *docker* apropriado apenas com base no cabeçalho "*Host*:" da requisição *web*.

```
location /sitexemplo {
    proxy_pass http://127.0.0.1:3000;
    # na porta 3000 do localhost,
    # um docker contendo a aplicação
    # "exemplo" é executada

}
```

Essa abordagem funciona bem, mas necessita que a aplicação seja feita em mente que ela "começa"em **"/algumnome>"** , o que nem sempre é verdade. A adaptação envolve a mudança de todos os links e todos os caminhos tanto no *front* quando no *backend* da aplicação. Em resumo, as aplicações que serão hospedadas no *hub* precisam para poderem operar:

- Ter um *Dockerfile* e *compose*
- "Iniciar" em **/algumnome**, sendo o nome de livre escolha
- Ter um repositório público. (*Github*, *Gitlab* etc)

Monitoramento e disponibilidade do *Hub*

O monitoramento no momento é feito apenas pelo *logs* gerados pelo *nginx*, *dockers*, *modsecurity* e *fail2ban*. Para visualização do uso de recursos, a ferramenta utilizada é o *htop*.

Quanto à disponibilidade, foi configurado uma tarefa que a cada 3 horas dispara 5 pacotes ICMP para o **google.com**. Caso esse teste falhe, provavelmente o servidor perdeu conexão com a *internet* por algum motivo, a tarefa irá *resetar* o servidor para tentar recolocar ele no ar. Isso é necessário, pois pode haver quedas de *internet* no campus.

```
#!/bin/bash

TMP_FILE=/root/server_health/alive
LOG_CHECKS=/root/server_health/log

no_inet_action() {
    shutdown -r +1 'No internet.'
}

if ping -4 -c5 google.com; then
    echo 1 > $TMP_FILE
    current_date_time=$(date +"%Y-%m-%d %H:%M:%S")
    echo "SERVER_ALIVE_AT: $current_date_time"
    >> $LOG_CHECKS
else
    current_date_time=$(date +"%Y-%m-%d %H:%M:%S")
    echo "SERVER_DEAD_AT: $current_date_time"
    >> $LOG_CHECKS
    [[ `cat $TMP_FILE` == 0 ]] && no_inet_action
    || echo 0 > $TMP_FILE
fi
```

Script anterior adaptado desse *post* no *StackExchange*

Em seguida, adiciona-se uma entrada no *cron*, o agendador de tarefas no *Linux*, para executar esse *script*:

```
crontab -e
# adicione a linha 0 */3 * * *
# /root/server_health/check_inet.sh
```

O *hub* também foi inscrito em um serviço de "*ping*" automático, que notifica por *e-mail* se o servidor não responder a alguma requisição. É possível ver o status do servidor em: status

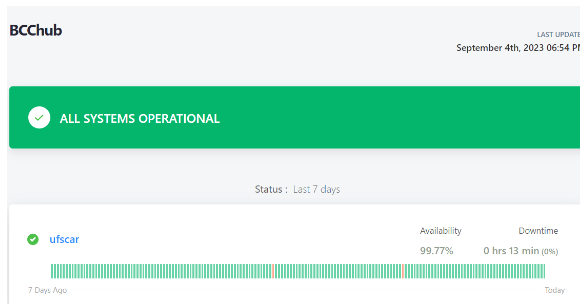

Figura 10.6: Página de *status* do **BCC-HUB**. Clique aqui para ver a imagem ampliada.

10.4 *Bugs*/problemas conhecidos

Atualmente, o *hub* tem problemas de disponibilidade devido a quedas/picos de energia no campus. Esse problema já ocasionou até a queima da fonte do servidor, que precisou ser trocada. A falta de *backup* do sistema é um problema também.

Além disso, é necessário encontrar alguma solução quanto a necessidade de adaptação dos sites para hospedagem. Idealmente, apenas o *Dockerfile* e *compose* seriam adicionados na aplicação, isso podendo ser feito por um administrador.

10.5 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

https://github.com/rafaelbarbeta/BCC-HUB