Módulo 1 - Linguagens de Alto Nível

Tarefa 1.1

Quais linguagens nos slides que você…
já tinha ouvido falar? em que situação?
COBOL, é uma linguagem utilizada em aplicações legadas do meu trabalho.
FORTRAN, meu irmão aprendeu na disciplina de programação para Engenharia.
LISP, utilizada por alguns colegas que já fizeram este curso de Estrutura de Linguagens.
Prolog, utilizei aqui na UERJ no curso de Inteligência Artificial.
C, utilizei aqui na UERJ no curso de Linguagem de Programação 1.
C++, utilizei aqui na uerj nos cursos de Estruturas de Dados 1 e 2.
Visual Basic, é uma linguagem utilizada em aplicações legadas do meu trabalho.
Java, é uma linguagem utilizada na maioria das aplicações do meu trabalho e também lecionada aqui na UERJ na disciplina de Linguagem de Programação 2.
PHP, por ter interagido com desenvolvedores web back-end.
Javascript, já utilizei para desenvolvimento web front-end.
Python, utilizo atualmente no meu trabalho.
Ruby, apenas escutei falar sobre essa linguagem, nunca interagi.
Lua, embora nunca tenha implementada nada com Lua, ja ouvi falar por ser bastante utilizada em desenvolvimento de Games.

já teve algum contato? em que nível?
COBOL, apenas leitura de alguns códigos.
FORTRAN, desenvolvimento de algoritmos básicos.
LISP, apenas leitura de alguns códigos.
Prolog, implementações na disciplina de Inteligência Artificial.
C, implementações na disciplina de Linguagem de Programação 1.
C++, implementações nas disciplinas de Estruturas de Dados 1 e 2.
Visual Basic, implementações no trabalho.
Java, implementações na disciplina de Linguagem de Programação 2.
PHP, pouquíssimo desenvolvimento.
Javascript, implementação no trabalho e em projetos pessoais.
Python, implementações no trabalho.

programa com frequência? em que contexto?
Programo com muita frequência em Python, pois trabalho na área de Big Data e também estou usando na implementação do projeto final.
Javascript é a segunda linguagem que mais utilizo, por implementações no trabalho e em projetos pessoais.
As demais, tive mais contato em disciplinas da UERJ ou mesmo pouca frequência de implementação.

mais teria curiosidade de conhecer? por quê?
Das linguagens apresentadas no slide, a que tenho mais curiosidade é Lua, por ouvir dizer que é amplamente utilizada na área de desenvolvimento de Games.

Embora não estejam listadas nos slides, outras duas linguagens que tenho curiosidade de aprender são Go, pela vasta utilização em aplicações nativas de nuvem e também Swift, utilizada no desenvolvimento de aplicativos para o sistema iOS.

Tarefa 1.2

Dentre as linguagens que você já usou...
qual você mais gosta? por quê?
Python, pela simplicidade, facilidade de aprendizado e abstração de vários "pormenores".

qual você menos gosta? por quê?
Java, por ser muito verboso e pela complexidade de questões que envolvem polimorfismo, heranças, etc.

Tarefa 2.1

Escolha uma linguagem de sua preferência…
Sobre portabilidade…
em que sistemas operacionais ela está disponível?
O JavaScript está disponível na maioria dos SOs populares como Linux e suas distribuições, Windows, MacOS e também em sistemas de dispositivos móveis. A Linguagem é exceção quando trata-se de Sistemas Operacionais de nicho, que atendem a algum hardware específico, como, por exemplo, sistemas embarcados, sistemas de IoT ou sistemas voltados à automação industrial.

em que arquiteturas ela está disponível?
Considerando a ressalva feita na resposta anterior, o JS está disponível em qualquer arquitetura. Por ser uma linguagem concebida para execução do lado cliente, nos navegadores de internet, o JavaScript depende de um interpretador disponível nas aplicações de browser. Isso significa que os detalhes de implementação referente a arquitetura do computador em que o código JS é executado, fica abstraído para o desenvolvedor.

em que sistema ou arquitetura ela não está disponível?
Sistemas Operacionais de nicho, que atendem a algum hardware específico, como, por exemplo, sistemas embarcados, sistemas de IoT ou sistemas voltados à automação industrial. Exemplos: TinyOS, um sistema operacional projetado para dispositivos de redes de sensores sem fio; Cisco IOS, sistema utilizado nos roteadores de rede da Cisco; OpenWrt, uma distribuição Linux embarcada em dispositivos wireless.

Sobre abstrações de dados…
que abstrações diferentes/especiais ela possui?
JavaScript é uma linguagem de tipagem dinâmica, isto é, não é necessário declarar o tipo de uma variável antes de sua atribuição, sendo esse tipo automaticamente determinado em

tempo de processamento. Além disso, no JS é possível re-atribuir uma mesma variável a um tipo diferente de dado. Devido a essa característica, uma variável no JavaScript pode receber os valores “null” ou mesmo “undefined”.
Objetos em JavaScript podem ser definidos através de uma notação JSON (Java Script Object Notation) que é um estrutura de dados permutável onde os dados são armazenados em pares tipo “chave-valor”.

que abstrações comuns ela não possui?
O JavaScript é uma linguagem de tipagem dinâmica e não possui structs e ponteiros, por exemplo. Conforme citado na resposta anterior, são utilizados objetos que armazenam atributos em notação JSON e são instanciados na memória podendo ser referenciados por atribuição de variáveis. Uma consideração importante é que no JS os arrays são objetos e podem ser manipulados como tal (com métodos de interação e modificação definidos). Não existe uma visão de vetor como em C, por exemplo, onde o conteúdo de um vetor é armazenado em região contígua de memória, limitada por dado tamanho ou que necessite de alocação dinâmica.

Sobre abstrações de controle…
que abstrações diferentes/especiais ela possui?
O JavaScript possui uma estrutura de repetição chamada “do while”. Essa instrução repete um bloco de comandos até que uma condição seja satisfeita. Embora seja similar ao comando “while” tradicional, diferente deste, o comando “do while” executa primeiro o conjunto de comandos internos e na sequência verifica a condição. Já no While, esse processo é invertido: primeiro verifica-se a condição e posteriormente são executados os comandos. Outras abstrações interessantes presentes no JS são os comando de controle “break” e “continue”. O “break” é utilizado para sair imediatamente de um laço, transferindo o controle para a próxima instrução. O “continue” pode ser usado para reiniciar uma instrução de repetição, mantendo a execução do laço a partir da próxima iteração.

que abstrações comuns ela não possui?
Não identifiquei ou não conheço nenhuma abstração de controle comumente utilizada em linguagens de programação que o JavaScript não possua.

Tarefa 2.2

Escolha uma linguagem de domínio específico de sua preferência…
Como ela ajuda na produtividade do seu domínio de aplicação?

Linguagem escolhida: GraphQL
O GraphQL é uma linguagem de consulta à APIs criada pelo Facebook em 2012 e lançada publicamente em 2015. Com o advento da web, tornaram-se muito comuns as arquiteturas REST para fornecer interoperabilidade entre serviços web. Esse modelo de arquitetura se baseia no uso do protocolo HTTP (e de suas operações básicas de GET, POST e DELETE, por exemplo) para tornar possível a integração de sistemas da internet através de APIs. Contudo, existe um problema relacionado à produtividade quando utilizamos qualquer linguagem de programação para consultar dados em um endpoint REST: ao usar uma API REST para buscar informações, você sempre receberá como resposta um conjunto de

dados completo, em geral no formato JSON, não sendo possível limitar os campos que essa API retorna; Esse fenômeno é conhecido como “over fetching” (quando você obtém mais dados do que realmente precisa). O GraphQL usa sua linguagem de consulta para adaptar a solicitação exatamente ao que você precisa, desde vários objetos até campos específicos dentro de cada entidade.
Um exemplo que pode ilustrar bem essa situação é uma consulta a API pública do GitHub, através da URL: https://api.github.com/events (é possivel ver o retorno acessarndo a URL na barra de endereços do browser).
Esse endpoint responde um conteúdo JSON bastante extenso, com várias hierarquias entre entidades. Isso significa que em uma chamada utilizando a linguagem Python, por exemplo, todo o objeto seria retornado, fazendo-se necessário a execução de um parse no conteúdo do JSON a fim de extrair apenas os campos desejados.
O JSON citado retorna alguns eventos (ex: fork, push e create) que ocorreram em repositórios públicos do github, no instante em que a chamada ao endpoint é realizada. Suponha que, no objeto retornado, desejamos obter apenas o campo “type” (tipo do evento) com valor “CreateEvent” e o campo “actor”, restringindo “actor” aos nós-filho “id” e “display_login”. Em Python a implementação ficaria da seguinte forma:

```
import requests
import json

url = 'https://api.github.com/events'
response = requests.get(url)
jsonfile = response.json()

response_filter = []

for field in jsonfile:
  if field['type'] == 'CreateEvent':
    keys = {
      "type":field['type'],
      "actor":{
        "id":field['actor']['id'],
        "display_login":field['actor']['display_login']
      }
    }
    response_filter.append(keys)

print(json.dumps(response_filter, indent=2))
```

Usando a linguagem GraphQL para consulta, a implementação ficaria da seguinte forma:

```
query {
    type (args: “CreateEvent”)
    actor {
        id
```

```
      display_login
    }
}
```

Essa consulta em GraphQL filtra todos os campos do JSON com chave “type” cujo valor é “CreateEvent” e todos os campos com chave “id” e “display_login”, aninhados ao campo cuja chave é “actor”, com uma sintaxe simples e objetiva, sem necessidade de parse no objeto retornado pela API.
Em contrapartida, o programa anterior, implementado em linguagem Python, fica mais complexo devido a necessidade de manipulação do JSON para filtrar logicamente o conteúdo e construir um novo objeto JSON contendo apenas os campos desejados.

Tarefa 3.1

Explique com suas próprias palavras o que você entende por semântica no contexto de linguagens de programação.
A semântica no contexto de linguagens de programação é o significado lógico das regras de execução de um programa. Ela está ligada a coesão da linguagem, isto é, ao sentido que uma construção de código, sintaticamente correto, denota. A semântica também se associa a abstração de detalhes de implementação que fazem das linguagens de programação de alto nível, mais próximas a linguagem natural (humana) e mais distantes da linguagem de máquina (instruções de hardware). Por exemplo: é mais fácil para um programador entender o sentido de uma instrução “for (int i = 0; i < 10 ; i++)” ou “for i in range(0,10)” do que instruções em bits (zeros e uns) de movimentação e carga entre registradores da máquina. Abaixo, está um segundo exemplo do contexto semântico de um trecho de código em python:

```
numbers = [2,4,6,8,10]
for number in numbers:
  continue
```

Sintaticamente o código está correto, contudo, semanticamente, ele contém um nível de abstração, mas não possui coesão. Esse trecho de código python cria uma lista de 5 números pares e atribui à variável chamada “numbers”. A abstração permite perceber que o comando for percorre cada elemento da lista, representado pela variável “number”. Entretanto, não existe um sentido lógico/semântico nesse contexto, pois esse programa simplesmente será executado sem nenhum objetivo concreto.

Tarefa 3.2

Dê exemplos de abstrações que escondem detalhes da entrada, saída, memória e CPU.

1. Abstração de entrada usando a linguagem Python:
```
import bluetooth
```

devices = bluetooth.discover_devices() #recebe sinais de entrada de dispositivos bluetooth ativos.

2. Abstração de saída usando a linguagem Python:
import simpleaudio as sa
wave_file = sa.WaveObject.from_wave_file(path_to_file) #abre e carrega um arquivo de áudio formato “wave”.
play_sound = wave_file.play() #executa o áudio nos dispositivos de saída de som.
play_sound.wait_done() #aguarda finalização da execução.

3. Abstração de memória usando a linguagem C:
int * pt; //declaração do ponteiro para armazenar o endereço do bloco alocado.
pt = (int *) malloc(10*sizeof(int)); //chamada da função malloc para reservar na memória espaço para 10 elementos do tipo int (4 bytes).
free(pt); //chama a função free para liberar os endereços ocupados na memória.

4. Abstração de detalhes da CPU usando a linguagem C:
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig); //aborta a execução de um processo escalonado pelo SO para execução na CPU, usando como parâmetro o pid (identificador do processo).

Tarefa 3.3

Descreva a sintaxe e semântica do comando for de C.

1. Sintaxe:

Na linguagem C, esse comando é escrito da seguinte maneira:
primeiro é descrita a instrução: “for”, seguida de um conjunto de configurações entre parênteses. Essas configurações são: uma atribuição inicial, um comando de comparação e um comando de atribuição final separados por ponto e vírgula. O conjunto de códigos a ser repetidos durante a execução da instrução, é indicados entre chaves:

```
for(atribuição inicial; condição de parada da repetição; atribuição final) {
        bloco de códigos executados durante a repetição;
}
```

Exemplo:

```
for(int i = 0; i <= 10; i++) {
        printf("%d ", i);
}
```

2. Semântica:

O comando for é uma estrutura de repetição onde é executado um conjunto de instruções enquanto uma determinada condição é verdadeira. Este comando utiliza uma variável de controle que é responsável pela contagem e incremento do loop, bem como a identificação da condição de parada.

```
for(int i = 0; i <= 10; i++) {
        printf("%d ", i);
}
```

for( ) : instrução que inicia a estrutura de repetição.
i = 0 : valor inicial atribuído a variável de controle.
i <= 10 : variável de controle que testa se a condição de parada é verdadeira. Em caso positivo, a repetição continua, caso contrário, a repetição é encerrada.
i++ : representa o passo de repetição, isto é, o incremento da variável de controle durante o loop.
{ : marca o início do bloco de comandos a ser repetido.
} : marca o fim do bloco de comandos a ser repetido.

Tarefa 3.4

Descreva a sintaxe e semântica de chamada de funções em Python.

1. Sintaxe:

Na linguagem Python, as funções são escritas da seguinte maneira:
primeiro é descrita a instrução “def”, seguida de seu nome e parâmetros, respectivamente. Uma função pode conter zero ou mais parâmetros que precisam ser escritos entre parênteses e são separados por vírgulas. Após os parâmetros, o cabeçalho da função é finalizado por dois pontos e o corpo da função (conjunto de comandos) é escrito na linha seguinte de forma indentada (tabulação). O nome da função não pode conter palavras reservadas, espaços, hífen ou acentos podendo ser composto por caracteres minúsculos, maiúsculos e/ou caracteres numéricos. Em Python, as funções podem ou não conter retorno que é marcado pela palavra reservada “return”. Exemplos:

```
def hello_world( ):
        print(“Hello World!”)

def sum(x, y):
        s = x + y
        return s
```

2. Semântica:

As funções em Python são conjunto de comandos que executam alguma tarefa específica. Em algumas outras linguagens são chamadas de funções as rotinas que retornam algum valor para o contexto de execução do programa. Caso esse retorno não exista, essas rotinas são chamadas de procedimentos. Contudo, em Python, semanticamente, não existe

essa separação, todas as rotinas, retornando ou não valores, são chamadas de funções. Portanto, além da sequência de comandos, as funções podem ou não retornar valores aplicáveis ao escopo global de execução do programa. As funções do Python podem ou não conter parâmetros que representam as variáveis utilizadas no escopo local de tal função. Quando uma função que necessita de parâmetros é utilizada, valores são atribuídos como argumentos de entrada que serão utilizados localmente durante sua execução.

Tarefa 3.5

Pesquise e descreva as principais diferenças semânticas entre Python 2 e Python 3.

DIFERENÇA 1: Função input() vs raw_input():
Em Python 2 existem duas funções que podem ser aplicadas na leitura de entradas fornecidas pelo usuário. Uma é input() e outra é raw_input(). Ambas funcionam nesta versão, mas tem finalidades diferentes. Observe os trechos de código abaixo:

```
>>> string_input = input(“Digite a entrada: ”)
Digite a entrada: entrada
  File “<input>”, line 1, in <module>
  File “<string>”, line 1, in <module>
NameError: name ‘entrada’ is not defined

>>> string_entrada = "entrada"
>>> string_input = input(“Digite a entrada: ”)
Digite a entrada: string_entrada
>>> print string_input
entrada
```

No primeiro caso o interpretador do Python 2 retorna uma “NameError”, pois a string fornecida como entrada não faz parte do contexto de execução do programa. Já no segundo trecho de código, o erro não ocorre pois a variável string_entrada inclui o valor atribuído ao contexto de execução. Uma forma de evitar a dependência de avaliação do contexto é utilizar a função raw_input(), que captura a entrada fornecida em formato de string e sem avaliação prévia.
Já no Python 3, a função raw_input() foi simplesmente renomeada para input(). Portanto, não existe essa diferenciação, apenas uma função pode ser executada para leitura das entradas e esta ocorre pela chamada input().

DIFERENÇA 2: Função Print()
No Python 2, print não é uma função, mas sim um statement, podendo ser aplicado da seguinte forma:

```
>>> print ‘Olá, mundo!’
‘Olá, mundo!’
```

No entanto, ao utilizar o Python 3, o comando print() torna-se, de fato, uma chamada de função, necessitando os parênteses para execução correta, caso contrário um erro de

sintaxe será lançado. Observe que no Python 3 ocorre um erro de sintaxe pela natureza da função, mas se comparado com o Python 2, a alteração de statement para função é uma diferença semântica, pois funções podem ser aplicadas de diversas formas ao longo do código, como por exemplo, sendo atribuída a variáveis e permitindo uso de vários argumentos; o que não é possível quando o print é usado em sua forma statement.

DIFERENÇA 3: Função range()
No Python 2, a função range() retorna uma lista de números inteiros, enquanto que no Python 3 range() passou a ser uma classe cujo argumento do construtor precisa ser um número inteiro. Isso significa que em Pyhton 3, para iterar o objeto é necessário utilizar a função list(). Exemplos:

Em Python 2:
```
>>> numeros = range(5)
>>> print numeros
[0, 1, 2, 3, 4]
>>> type(numeros)
<type ‘list’>
```

Em Python 3:
```
>>> numeros = range(5)
>>> print(numeros)
range(0, 5)
>>> type(numeros)
<type ‘class’>
>>> print(list(numeros))
[0, 1, 2, 3, 4]
```

DIFERENÇA 4: Divisão por número inteiro
No Python 2, toda divisão envolvendo dois números inteiros resulta em um terceiro número inteiro. Caso o resultado seja um número fracionado, será exibido o valor truncado. Em Python 3, o comportamento é diferente pois esse tipo de divisão resulta em um número float. Exemplo:

Em Python 2:
```
>> 4/2
2
>> 3/2
1
```

Em Python 3:
```
>> 4/2
2.0
>> 3/2
1.5
```

Tarefa 4.1

Qual é a diferença fundamental entre linguagens imperativas e linguagens funcionais?

As linguagens imperativas foram influenciadas pela arquitetura clássica de von Neumann e portanto projetadas para execução nesse modelo de máquina. Em um computador com essa arquitetura, tanto dados como programas são armazenados em memória e a CPU é a unidade responsável pela execução das instruções. Isso significa que é necessário estabelecer uma comunicação entre memória e CPU transferindo entre elas operandos e resultados. Nesse contexto as linguagens imperativas funcionam muito bem pois possuem como característica central a utilização de variáveis que guardam o estado atual do programa e correspondem ao endereço de localização na memória. Esse estado pode ser alterado através de comando de atribuição, o que introduz o conceito da execução sequencial e ordenada. Além disso as linguagens imperativas contam com estruturas de repetição que são utilizadas para varrer uma sequência de endereços na memória e/ou acumular valores em uma variável. Já as linguagens funcionais, baseiam-se no conceito matemático de funções que mapeiam um conjunto domínio em um conjunto imagem. Essas linguagens são formadas exclusivamente por funções onde os códigos são escritos de maneira declarativa através de expressões. Na abordagem funcional, o computador trabalha de forma semelhante a uma calculadora que avalia expressões escritas pelo programador através de simplificações até chegar a uma forma normal. Portanto, linguagens funcionais não estão diretamente ligadas ao estado de um programa, pois nelas não existe alteração de estados e sim transformação de coleções de dados, isto é, o controle de fluxo do programa ocorre através das chamadas de função incluindo a recursão.

Exemplos:

```
# Python imperativo
numbers = [1, 2, 3, 4, 5]
x = []
for num in numbers:
        x.append(num*2)
print(x)

# Python funcional
print(list(map(lambda x : x*2, [1, 2, 3, 4, 5])))
```

Referências:

1. LINGUAGENS IMPERATIVAS & LINGUAGENS FUNCIONAIS: http://www.ppgsc.ufrn.br/~rogerio/material_auxiliar/CLP20131_linguagens_imperativas_funcionais.pdf

2. PARADIGMA IMPERATIVO: https://www.inf.pucrs.br/~gustavo/disciplinas/pli/material/paradigmas-aula09.pdf

3. PARADIGMA FUNCIONAL
https://www.inf.pucrs.br/~gustavo/disciplinas/pli/material/paradigmas-aula15.pdf

4. Linguagens Imperativas:
https://www.ic.unicamp.br/~ra100621/class/2019.1/PL_files/curso/01-programacaoImperativa/programacaoImperativa.html

5. Programação Funcional — Guia de Introdução:
https://medium.com/true-henrique/programa%C3%A7%C3%A3o-funcional-pura-ruby-e-monads-i-introdu%C3%A7%C3%A3o-b16687db63d

6. Programação funcional versus programação imperativa:
https://docs.microsoft.com/pt-br/dotnet/standard/linq/functional-vs-imperative-programming

Tarefa 4.2

Na sua opinião, por quê o paradigma de orientação a objetos se tornou tão popular?

A Programação Orientada a Objetos, foi criada para tentar aproximar o mundo real e o virtual: a ideia fundamental é tentar simular nossa realidade durante a confecção do programa utilizando o conceito de classes e objetos. Um exemplo dessa simulação de objetos do mundo real sendo traduzidos em classes e objetos virtuais, que aproximam as duas realidades, é a forma de descrever um celular, por exemplo: existem diferentes tipos de telefones celulares que podem ter recursos diferentes, mas há um conjunto básico de atributos que é comum a qualquer celular: todos eles têm telas, todos têm um sistema operacional, todos fazem chamadas de algum tipo e a maioria envia mensagens de texto de alguma forma. O conjuntos base que define os atributos e métodos associados ao celular (como a realização das ligações e envio de mensagens) configura uma classe que instancia objetos, que por sua vez, possuem características específicas (como de um smartphone por exemplo). Na Programação Orientada a Objetos, o programador molda os objetos a interação entre eles. Isso possibilita a criação de códigos que podem ser facilmente reutilizados, pela flexibilidade de extensão das classes. Portanto, as linguagens de programação que funcionam dessa forma se tornam fáceis de visualizar e traduzir. Essa associação, possivelmente, é o principal motivo da popularização do paradigma OO. Contudo existe um segundo fator que historicamente contribuiu para tal difusão. O paradigma de Orientação a Objetos foi concebido durante a década de 60, entretanto, teve grande expansão e adoção a partir do fim da década de 80 e início da década de 90. Esse fato ocorreu principalmente porque os aspectos desse paradigma possibilitaram por exemplo o surgimento das primeiras interfaces gráficas dos computadores pessoais. A ascensão do sistema operacional Windows no início dos anos 90 é um bom exemplo. Antes de se tornar o famoso “Windows”, o MS-DOS (assim como outros sistemas) era baseado em linhas de comando em terminal, o que dificultou a popularização do uso de computadores, pois estava mais restrito aos conhecimentos técnicos. O uso das interfaces gráficas com janelas, facilitada pela flexibilidade das linguagens OO (conforme discutido anteriormente) permitiu que novas funcionalidades surgissem e com isso novos programas e softwares passaram a ser implementados, demandando a difusão do conhecimento em programação utilizando esse tipo de paradigma.

Tarefa 4.3

Explique com suas próprias palavras a função "eval" de Python 3.
A função eval() permite que, em tempo de execução, um programa escrito em Python execute um outro código também escrito em Python. Ela recebe uma string como argumento de entrada e avalia a expressão especificada, caso seja uma instrução Python válida, essa string será interpretada como código e executada. Caso contrário, uma exceção será retornada. Segue abaixo um pequeno exemplo onde a função eval() pode ser bem útil. Esse trecho de código utiliza um comando para retornar o diretório corrente/atual do usuário:

```
>>> import os
>>> eval('os.getcwd()')
'/home/bruno_criscuolo'
```

No exemplo descrito acima, a funcionalidade pode facilitar a criação de diretórios, durante a execução de um script, para armazenar alguns arquivos temporários ou de stage, por exemplo.

Tarefa 4.4

Dê exemplos de outras características de linguagens dinâmicas.

Tarefa 4.5

Escolha três áreas da computação (ex., computação científica, software básico, etc) e defenda o uso de uma linguagem para cada uma delas.

1. Computação Científica: Linguagem R
A Linguagem R foi desenvolvida com foco em modelagem estatística e se caracteriza por uma abordagem centrada em data frames, isto é, dados matriciais, que na prática são uma lista de vetores de igual comprimento. Essa característica facilita agrupamento e análise de grandes conjuntos de dados. R foi originalmente implementada usando bibliotecas da linguagem de programação Fortran, que é tradicionalmente aplicada na Computação Científica pela capacidade de execução dos algoritmos da Álgebra Linear Numérica. A Linguagem R possui diversas funções nativas para manipulação de matrizes e vetores de forma facilitada, não sendo necessárias grandes implementações ou uso de pacotes desenvolvidos por terceiros. Além disso, o R também apresenta uma série de recursos para plotagem de gráficos e objetos geométricos, como segmentos de retas, círculos, cubos, esferas, etc. Em resumo, R é uma linguagem que fornece ao programador alta capacidade para análise estatística e de dados, prototipação e experimentação numérica.

2. Internet / Web: Node JS

Node.js é uma linguagem orientada a eventos que permite processamento de requisições I/O (input e output) não-bloqueante, isto é, habilita a execução de tarefas simultâneas e independentes sem requisições sequenciais. Essa característica o torna leve e eficiente, ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos. O Node é capaz de processar dezenas de milhares de conexões simultâneas utilizando apenas uma única thread, pois, apresenta uma programação assíncrona que compartilha recursos em uma máquina virtual. Essa máquina virtual fica então responsável por verificar qual tarefa deve ser executada, delega a atividade e voltar à atender novas requisições enquanto esse processamento paralelo está acontece. Isso reduz fortemente a utilização de recursos do hardware, pois em sua grande maioria, os programas BackEnd passam a maior parte do tempo lendo e escrevendo no disco. Esses são os motivos pelos quais o Node.js tem sido muito aplicado no desenvolvimento server-side. Além de ser leve e multiplataforma, a linguagem permite ainda criar APIs que são facilmente escaláveis: uma vantagem no que diz respeito a integração de sistemas web.

3. Empresas e Negócios: Swift
O Swift foi criado pela Apple como uma derivação da linguagem Objective-C com uma sintaxes mais simples, que lembra lembra linguagens de script como Ruby e Python. O Swift foi criado com ideal de ser uma linguagem mais produtiva, onde o programador consegue fazer mais coisas do que faria tradicionalmente com Objective-C. As Classes, Estruturas e Protocolos do Swift tornaram-se construções mais flexíveis e de uso geral, é possível utilizar tipos de dados avançados como Tuples e funções aninhadas e com retorno múltiplo. Todas essas características demonstram o foco do Swift em produtividade e na abstração de alto nível para implementação das aplicações. Além disso, o Swift faz uso de um mecanismo chamado ARC (Automatic Reference Counting) que rastreia e gerencia o uso da memória pelo aplicativo, fazendo com que o programador, pelo menos na grande maioria dos casos, não tenha que se preocupar com essa importante questão, que pode derrubar a execução de um app.

Referências:

1. Conheça as 5 melhores linguagens de programação para inteligência artificial: https://computerworld.com.br/plataformas/conheca-5-melhores-linguagens-de-programacao-para-inteligencia-artificial/

2. O que é, onde aplicar e quais as vantagens da Linguagem R: http://micreiros.com/o-que-e-onde-aplicar-e-quais-as-vantagens-da-linguagem-r/

3. Node.js - O que é, por que usar e primeiros passos: https://medium.com/thdesenvolvedores/node-js-o-que-%C3%A9-por-que-usar-e-primeiros-passos-1118f771b889

4. POR QUE UTILIZAR NODE.JS? https://isitics.com/2019/06/04/por-que-utilizar-o-node-js/#:~:text=(open%20source).-,Node.,dados%20atrav%C3%A9s%20de%20dispositivos%20distribu%C3%ADdos.

5. Top 10 linguagens de programação mais usadas no mercado: https://www.devmedia.com.br/top-10-linguagens-de-programacao-mais-usadas-no-mercado/39635

6. Swift: a linguagem que aproxima o mundo da programação: https://imasters.com.br/back-end/swift-linguagem-que-aproxima-o-mundo-da-programacao

Tarefa 4.6

Leia o artigo "Beating the Averages" de Paul Graham e explique com suas próprias palavras o "Paradoxo de Blub".
O "Paradoxo de Blub" é descrito pelo autor como uma situação na qual um programador usa como parâmetro comparativo a linguagem que domina. Isso o leva a considerar linguagens de programação menos poderosas do que aquelas que ele conhece como carentes de recursos importantes, ao passo que linguagens mais poderosas tornam-se desnecessárias ou complexas. Para ilustrar esse comportamento, Paul Graham, faz uso de uma linguagem de programação hipotética chamada Blub considerada intermediária num espectro das linguagens de baixo até alto nível. Paul afirma que enquanto um programador de Blub estiver olhando para linguagens menos poderosas, ele as tomará como incompletas, cuja implementação necessita de muita programação extra por falta de uma funcionalidade X contemplada em Blub. Mas quando esse mesmo programador olha na direção oposta, ele não percebe que existem linguagens mais poderosas. Para esse programador, existem linguagens consideradas equivalentes em poder a Blub, mas que contém funcionalidades estranhas. Logo, Blub é bom o suficiente para ele, porque ele pensa em Blub. Em contrapartida, um programador de uma linguagem de nível mais alto enxerga Blub como incompleta por não ter funcionalidade Y. Portanto, o "Paradoxo de Blub" nos leva a concluir que se um programador está condicionado à linguagem que usa, essa linguagem ditará a forma como ele pensa em termos de programação.

Tarefa 5.1

Descreva em linhas gerais um trabalho ou projeto que você tenha participado… Quais seriam os 2 critérios *externos* mais importantes para esse projeto? Justifique. Qual linguagem foi usada no projeto e qual seria a melhor? Justifique.

Participei de um projeto na empresa onde trabalho cujo objetivo era fazer uma ingestão de dados de monitoramento de condução veicular que eram transmitidos por um aplicativo em tempo quase real. Os dados de frenagem, latitude, longitude, velocidade eram transmitidos minuto a minuto para uma plataforma de Big Data de uma empresa parceira e essa encaminhava um lote de arquivos, com cerca de 2GB cada, contendo os dados para carga em nosso Data Lake e posterior organização no Warehouse. Para esse projeto, foram analisados diversos critérios e foi decidido utilizar a Linguagem Python aplicando o framework Apache Beam. Os critérios externos levados em consideração foram a portabilidade e desempenho. A portabilidade em virtude de o framework da Apache ser open source e python ser uma linguagem altamente portável em função da aplicabilidade e

difusão nas diversas plataformas para processamento de dados em nuvem. Foi levado em conta também o desempenho visto que são milhares de clientes usando o aplicativo com volume alto de dados sendo transmitido em curto intervalo de tempo. Nesse cenário o Apache Beam aplicado junto ao Python permitiu processar os arquivos de forma distribuída utilizando a abordagem funcional da linguagem, de forma que os dados podiam ser processados em modelo pipeline. Assim, os dados eram rapidamente recebidos processados e disponibilizados para análise na camada do Warehouse.
Existia também a possibilidade de usar o Java junto ao Apache Beam, porém, acredito que a escolha do Python foi assertiva devido há outros dois critérios: a agilidade no desenvolvimento e também a facilidade de manutenção, dado que ao fim do projeto a solução seria entregue para outra equipe operar.

Tarefa 5.2

Descreva em linhas gerais um trabalho ou projeto que você tenha participado… Quais seriam os 2 critérios *internos* mais importantes para esse projeto? Justifique. Qual linguagem foi usada no projeto e qual seria a melhor? Justifique.
Na empresa onde trabalho, participei de um projeto cujo objetivo foi desenvolver processos de monitoramento de consulta realizadas no Data Warehouse visando especialmente a segurança da informação. O propósito geral foi construir alguns programas em Python capazes de monitorar o log da plataforma a partir de alguns critérios como: volume em bytes retornados por consultas, quantidade de linhas consultas, horário de execução, entre outros. Esses programas de monitoramento eram responsáveis pela verificação da mensagem de log, processamento dos critérios e regras estabelecidas e posterior carga em um conjunto restrito de tabelas do próprio Warehouse que conjugavam uma fonte de dados para painéis analíticos. Além disso, dependendo da classificação de risco capturada e classificada nas mensagens, o programa Python ficava responsável também por disparar um alerta via e-mail, direcionado à um grupo restrito de pessoas responsáveis pelas operações de contensão. Nesse caso, acredito que os critérios internos mais importantes foram a legibilidade e redigibilidade, pois em termos de manutenção era necessário que as regras aplicadas no monitoramento ficassem claras e acessíveis, bem como novas regras que, porventura, fossem adicionadas como evolução do projeto. Penso que o Python foi uma boa escolha pois além dos motivos citados anteriormente, a linguagem permite abstração de dados para manipulação de arquivos em formato JSON, por exemplo, o que possibilitou uma integração facilitada com as mensagens dos eventos de log e com disparos de e-mail via API.

Tarefa 5.3

Escolha uma linguagem de sua preferência. Dê um exemplo de inconsistência ou não ortogonalidade entre funcionalidades da linguagem.
Um exemplo de não ortogonalidade entre funcionalidades da linguagem Python 3 pode ser expresso pelo uso das funções input() e append(). Segue abaixo um exemplo com um trecho de código onde é criada uma função responsável por somar elementos de uma lista.

Nesse caso, o código imprimirá na tela o valor 15, como resultado do somatório dos elementos contidos na lista.

```
1  def soma_elementos(vetor):
2      soma = 0
3      for elemento in vetor:
4          soma = soma + elemento
5      return soma
6
7  lista = [0,1,2,3,4,5]
8  print(soma_elementos(lista))
```

Agora suponha que desejamos acrescentar na lista elementos que possam ser fornecidos pelo usuário, com objetivo de incorporar novos valores ao somatório. Para isso, foram adicionadas as linhas 8 e 9 no código (escrito abaixo), responsáveis, respectivamente por capturar a entrada fornecida pelo usuário e acrescentar um elemento no fim da lista. Porém esse código se torna inconsistente, uma vez que o retorno da função input() é uma string, inviabilizando a operação matemática de soma dos elementos da lista, realizada pela função “soma_elementos”. Como Python é uma linguagem dinamicamente tipada, a lista “declarada” não possui um tipo específico, o que possibilita a inclusão de elementos de qualquer tipo (nesse caso, inteiros e strings). Porém quando aplicada uma operação aritmética utilizando essa lista que contém “mesclagem de tipos”, um erro é retornado como exceção.

```
1  def soma_elementos(vetor):
2      soma = 0
3      for elemento in vetor:
4          soma = soma + elemento
5      return soma
6
7  lista = [0,1,2,3,4,5]
8  novo_elemento = input("Entre com um valor: ")
9  lista.append(novo_elemento)
10 print(soma_elementos(lista))
```

Tarefa 5.4

Pesquise dois códigos que façam a mesma coisa, mas em linguagens diferentes, e discuta sobre a legibilidade e redigibilidade de ambos.

Ambos os códigos apresentados abaixo executam recursivamente uma função para calcular o fatorial de um número recebido como parâmetro: um dos códigos está escrito em LISP e outro em VB. Nota-se que a legibilidade do código escrito em Visual Basic é muito mais agradável do que a do código escrito na linguagem LISP. Podemos considerar alguns aspectos que favorecem a legibilidade do código VB. Em primeiro lugar, é possível observar que em VB existe uma tipagem claramente declarada para a variável “n” (Integer) e também para a função “Factorial” (declarada como tipo Long, que pode retornar valores inteiros de até 8 bytes). Nesse programa é fácil compreender também as expressões condicionais,

onde iniciam, onde terminam e o que é feito quando existe o desvio condicional. Observamos todos os blocos do programa bem demarcados pelos statements “End If” e “End Function”, por exemplo. Também podemos destacar os retornos da função e a sintaxe clara do cabeçalho da função. Todas essas características convergem para uma boa legibilidade do código VB, o que o torna facilmente entendível por qualquer programador, mesmo não acostumado com a linguagem, favorecendo também a manutenção do código. Quando analisamos o código LISP, em relação a legibilidade, fica aparente a dificuldade de interpretação da função. Isso acontece porque no programa LISP estão abstraídas uma série de declarações que, se comparadas ao VB, tornam esse código, à primeira vista, mais complexo. Por exemplo: qual é o tipo da variável “n” utilizada na função? Percebemos que existe uma condição demarcada pela palavra reservada “if”, mas o que acontece quando ocorre um desvio condicional? Existe uma condição “else” clara para esse “if”? Onde essas condições são finalizadas? O que a função retorna? A sintaxe de operadores e operandos também parece confusa. Pelo fato de a linguagem LISP utilizar o paradigma funcional, a sintaxe do código lembra algumas associações matemáticas e por isso existe uma série de demarcações por parênteses. Alguns elementos do código ficam mais claros como a variável “n”, o cabeçalho da função demarcado por “defun” (define function) seguido do nome da função (“fatorial”). Porém, a legibilidade do LISP fica mais restrita a quem entende da linguagem ou mesmo à um programador específico que redigiu esse código-fonte, dificultando a manutenção desse programa. Portanto, o exemplo abaixo comprova a pouca legibilidade do LISP frente ao código VB para função fatorial. Em termos de redigibilidade, o cenário se inverte, pois, embora o LISP exija uma sequência de parênteses para demarcar a função e seus blocos, é possível escrever toda a função fatorial em apenas uma linha, com declarações curtas e pouco verbosas. O mesmo não ocorre com o VB: esse código é escrito em várias linhas que contém expressões lexicais muito claras e bem estabelecidas que necessitam de escrita detalhada, a exemplo do “As” na tipagem das variáveis e da função. Essa estruturação do VB se aproxima um pouco mais da linguagem natural e até mesmo do pseudocódigo se comparado ao código LISP, cujas expressões são caracteristicamente mais matemáticas. Por fim, podemos concluir que, relativo à redigibilidade, o LISP é, na prática, mais redigível do que o código Visual Basic.

Fatorial escrito na linguagem LISP:

```
(defun fatorial (n) (if (<= n 1) 1 (* n (fatorial (- n 1)))))
```

Fatorial escrito na linguagem Visual Basic:

```
Function Factorial(n As Integer) As Long
    If n <= 1 Then
        Return 1
    Else
        Return n * Factorial(n - 1)
    End If
End Function
```

Tarefa 6.1

Sobre a frase de Alan Perlis a seguir...

"A language that doesn't affect the way you think about programming, is not worth knowing." (Uma linguagem que não afeta a maneira como você pensa sobre programação, não vale a pena conhecer.)
Você concorda ou discorda da frase? Por quê?
Qual a relação entre essa frase e a discussão sobre expressividade em linguagens de programação?
Sim, concordo. Porque uma só linguagem prende o raciocínio do programador à único paradigma por exemplo. Conforme foi estudado, é uma situação típica para o “Paradoxo de Blub”. Isso significa que o programador que fixa sua forma de pensar sobre a programação não terá versatilidade para resolução de diferentes problemas nos quais uma ou outra linguagem pode se destacar, ficando preso a sua forma de pensar e consequentemente a uma diversidade restrita de linguagens.
Com relação a discussão de expressividade a frase demonstra que quanto mais expressiva é uma linguagem mais ela tem influencia e afeta a maneira como se pensa a respeito da programação. Isso acontece pois a expressividade está ligada ao quanto uma linguagem mais expressiva necessita de reescrita para ser “traduzida” em uma linguagem menos expressiva e isso implica diretamente na forma pensar do programador e na maneira como será estruturada sua lógica de programação.

Trabalho

Tarefa 1.1

Qual linguagem você escolheu? Por quê?
A linguagem escolhida para o trabalho foi NodeJS.
O trabalho será desenvolvido em dupla com o Daniel Calábria e o motivo pelo qual escolhemos a linguagem foi o fato de estar sendo amplamente aplicada no mercado com objetivo de integrar sistemas web. Portanto, a intenção é entender um pouco melhor sobre o NodeJS, aprender sobre suas funcionalidades e também ter esse conhecimento mais alinhado com uma realidade voltada para web, que é uma área pouco abordada no currículo atual do nosso curso na UERJ.

Tarefa 1.2

Sobre a funcionalidade de alta expressividade você escolheu…
Qual é a funcionalidade?
A funcionalidade escolhida é Async/Await no contexto de programação assíncrona no NodeJS.
Como você chegou até ela?
Estudando um pouco mais sobre a Linguagem. Dessa forma, percebi que a principal vantagem do NodeJS é trabalhar de forma assíncrona no atendimento de requisições, evitando bloqueio por I/O e otimizando o consumo de hardware. Para entender pq a linguagem é capaz de fazer isso, cheguei aos callbacks assíncronos que consequentemente levaram até a funcionalidade Async/Await.

Você já tinha ouvido falar dela?
Já havia escutado sobre callbacks que são base fundamental para o entendimento de Async/Await. Ainda assim, o contato que tive com callbacks foi mais voltado para programação síncrona. Portanto, ainda não tinha escutado ou programado utilizando essa funcionalidade.
Você teve alguma dificuldade em entedê-la conceitualmente? Por quê?
Sim, bastante dificuldade. O principal motivo que dificulta o entendimento da funcionalidade é o fato de ela depender de alguns conceitos da programação assíncrona que não é trivial de ser entendida a primeira vista, dado que tradicionalmente é de maior costume programar de forma síncrona, isto é, pensando na execução de instruções sequenciais, linha a linha.
Por quê você escolheu ela?
Pois além de ser uma funcionalidade desafiadora que agregará conhecimento na forma de pensar sobre outros métodos de programação, essa funcionalidade é o "core" do motor do NodeJS. É graças ao aspecto assíncrono que o NodeJS tem alta capacidade de processamento das requisições e por isso tem sido bastante adotado para integração de sistemas web.

Módulo 2 - Amarração de Nomes

Tarefa 1.1

Sobre identificadores em LISP (e/ou derivados)…
Quais caracteres podem ser usados?
Em linguagens como LISP e derivados podem ser utilizados uma diversidade ampla de caracteres como identificadores de nomes (a maioria, incluindo acentos, por exemplo). Não podem ser usados: parênteses, colchetes, chaves, ponto, dois pontos, ponto e vírgula, crase, tralha, e comercial, pipe, aspas simples ou aspas dupla.
Por quê não são permitidos em outras linguagens?
Pois tratam-se de palavras reservadas que definem as regras de sintaxe e estrutura da linguagem. Elas não podem ser usadas como nomes de variáveis pois os compiladores ou interpretadores dessas linguagens utilizam essas regras sintáticas para gerar o respectivo código de máquina ou bytecode que serão executados pelo hardware. Nesse caso é necessário saber quando ocorrem, em cada instrução as operações lógicas e aritméticas, desvios condicionais e loops, por exemplo. LISP e sua família de linguagens, ao contrário das outras, é orientada a expressões, isto é, todos os códigos e dados são escritos como expressões em funções. As expressões são compostas por operadores e operandos que são identificados sintaticamente pela ordem em que aparecem nas declarações. O elemento mais à esquerda refere-se ao operador e os restantes dos símbolos são interpretados como nomes de variáveis. Quando uma expressão é avaliada pelo interpretador, ela precisa estar de acordo com a regra sintática, que caso esteja correta pode então produzir um valor que, por sua vez, pode ser embutido em outras expressões.
Que usos incomuns e interessantes você encontrou?
Na linguagem Scheme é possível vincular um nome a qualquer valor utilizando o operador "define". Ele declara o primeiro argumento como um parâmetro global e o vincula ao segundo argumento, mas isso pode gerar alguns problemas como no exemplo de código abaixo:

```
> (define + -)
> (+ 1 1)
0
```

Na primeira linha do código acima, o operador “define” vincula o símbolo de subtração (“-”) a variável global cujo nome é o símbolo de adição (“+”). A execução da segunda linha, faz o código retornar o valor zero. Isso ocorre porque o nome “-” foi vinculado ao símbolo da operação de soma que está mais à esquerda na expressão. Portanto o código é interpretado como uma operação de subtração e não de soma. Se a primeira linha de código fosse excluída, o valor esperado no retorno da operação deveria ser 2.

Tarefa 1.2

Sobre átomos ou símbolos em LISP, Erlang e Ruby…
o que são?
Átomos são os elementos básicos de uma expressão em linguagens funcionais. Um átomo pode possuir três tipos: numérico, string ou símbolo. Quando um átomo do tipo número ou string é avaliado, ele retorna o próprio valor, em outras palavras, a avaliação de um número resulta no próprio número, e o mesmo ocorre com as strings. Os números são bastante simples: qualquer sequência de dígitos - possivelmente precedida de um sinal (+ou -), contendo um ponto decimal (.) ou uma barra indicando frações (/). Strings são uma sequência de caracteres (que podem ser alphanumericos) colocados entre aspas duplas. Já os símbolos são um tipo de átomo com comportamento um pouco diferentes: trata-se de nomes que são utilizados para representar um dado referenciado. A avaliação de um símbolo resulta no seu valor como dado. Símbolos não precisam ser explicitamente criados. A primeira vez que o interpretador vê um símbolo, ele o cria automaticamente.
para que servem?
Os átomos são elementos responsáveis por constituir a estrutura da expressão funcional. O Átomo “nil” representa o valor nulo. Exemplo de átomos: a, foo, bar, 1, 0, nil. Já os símbolos servem para vincular um valor ou dado a um nome. Por exemplo, na expressão (setq x 8) existem 3 átomos: setq, x e 8. O primeiro átomo é um símbolo que serve para identificar uma função responsável por vincular o segundo átomo x - que também é um símbolo - ao terceiro átomo numérico, que é o número inteiro oito. Um segundo exemplo similar é (setq y “lisp”), neste caso o terceiro átomo da expressão é uma string que é vinculada ao símbolo y (segundo átomo).
que usos interessantes você encontrou?
As listas são um exemplo interessante, pois são formadas por uma um sequência ou coleção de objetos arbitrários em uma expressão. Por exemplo:

```
(); é uma lista com um átomo nil ou lista vazia
(1 2 3) ; é uma lista com 3 átomos numéricos
("foo" "bar"); uma lista de duas strings
(x 1 "foo"); uma lista de um símbolo, um número e uma string
```

Agora, vamos observar essa lista:

(+ (* 2 3) 4); uma lista de um símbolo, uma outra lista e um número.

Mas ao mesmo tempo essa lista contém símbolos que representam as operações de soma e multiplicação. Portanto trata-se também de uma expressão, cujo primeiro átomo é um símbolo vinculado a operação de soma e o segundo átomo é uma lista, que por sua vez contém um primeiro átomo que também é um símbolo, mas este vinculado a operação de multiplicação. Se essa expressão for avaliada e executada, apresentará o valor 10 como resultado.

De forma similar, podemos observar um outro exemplo interessante: uma lista de quatro átomos que contém dois símbolos, a lista vazia e outra lista, ela mesma contendo um símbolo e uma string:

(defun hello-world () (print "olá, mundo"))

Dessa forma, a lista acima com todo o conjunto de átomos e símbolos, na prática representa uma expressão que é uma função cuja finalidade é imprimir na tela um “olá, mundo”.

Um último exemplo interessante de funções utilizando símbolos é o operador especial (quote expressão) que retorna a expressão diretamente, sem tentar qualquer forma da avaliação. Por exemplo: (quote lisp) retorna lisp, e (quote (aprendendo lisp)) retorna (aprendendo lisp). O operador “quote” pode ser escrito na forma sintática de aspas simples (‘), isto é, no lugar de escrever (quote (+ 1 2)), por exemplo, você pode escrever '(+ 1 2). Ambas as expressões terão a mesma aparência: uma lista cujo primeiro elemento é o símbolo quote e cujo segundo elemento é a lista (+ 1 2).

Tarefa 1.3

Sobre os prefixos em variáveis Perl...
quais são e o que representam?
A linguagem Perl utiliza três tipos de prefixos para identificar tipos de variáveis diferentes:
$ = Indicam valores escalares que podem ser strings, inteiros ou números de ponto flutuante. Os valores escalares são sempre nomeados com '$', mesmo quando se referem a um escalar que faz parte de um array ou hash.
@ = Usado para denotar variáveis do tipo array ou matrizes. Ao utilizar a notação @array onde Perl espera encontrar um valor escalar ("no contexto escalar"), a linguagem fornecerá o número de elementos no vetor (tamanho do array).
% = Indica variáveis do tipo hash. Um hash representa um conjunto de pares de chave / valor.
quais são as 6 regras de conversão existentes entre eles? (ex., $xxx = @yyy)
1. Conversão $escalar = @array
Quando uma variável denotada por @array é atribuída à uma variável escalar escrita na forma $escalar, o Perl identifica o contexto dessas variáveis e atribui à $escalar apenas o número de elementos neste array, assumindo que o contexto é escalar. Exemplo:

@array = (1, 2, 3);

```
$escalar = @array;
print “Resultado: $escalar”
```

Output:
Resultado: 3

2. Conversão $escalar = %hash
Quando uma variável tipo hash, com notação %hash é atribuída a uma variável escalar com sintaxe $escalar, o Perl se comporta de forma similar ao caso anterior, com arrays, atribuindo à variável escalar o tamanho da hash, isto é, a sua quantidade de pares chave valor. Exemplo:

```
%hash = (
   'Primeiro' => 1,
   'Segundo' => 2,
   'Terceiro' => 3
);
$escalar = %hash;
print "Resultado: $escalar";
```

Output:
Resultado: 3

3. Conversão @array = $escalar
Quando uma variável do tipo array recebe um valor escalar, o Perl assume que o vetor passa a ser de único elemento, onde o valor na posição 0 corresponde ao valor escalar atribuído anteriormente. Exemplo:

```
@array = (1, 2, 3);
print "Antes: @array\n";
$escalar = 4;
@array = $escalar;
print "Depois: @array\n";
push(@array, 5);
print "Novo Array: @array";
```

Output:
Antes: 1 2 3
Depois: 4
Novo Array: 4 5

4. Conversão @array = %hash
Quando uma variável tipo hash é atribuída a um vetor, o Perl atribui à @array um vetor contendo os pares de elementos da hash (chaves e valores) em ordem arbitrária. Note que, no exemplo abaixo os valores sempre são precedidos de suas chaves nas posições do array, contudo, em execuções diferentes, os pares podem alterar as posições no vetor.

```
%hash = (
```

```
   'Primeiro' => 1,
   'Segundo' => 2,
   'Terceiro' => 3
);
@array = %hash;
print "Resultado: @array\n\n";

print "Indice 0: @array[0]\n";
print "Indice 1: @array[1]\n";
print "Indice 2: @array[2]\n";
print "Indice 3: @array[3]\n";
print "Indice 4: @array[4]\n";
print "Indice 5: @array[5]\n";
```

Output 1 (primeira execução do programa):
Resultado: Terceiro 3 Primeiro 1 Segundo 2
Indice 0: Terceiro
Indice 1: 3
Indice 2: Primeiro
Indice 3: 1
Indice 4: Segundo
Indice 5: 2

Output 2 (segunda execução do mesmo programa):
Resultado: Primeiro 1 Terceiro 3 Segundo 2
Indice 0: Primeiro
Indice 1: 1
Indice 2: Terceiro
Indice 3: 3
Indice 4: Segundo
Indice 5: 2

Output 3 (terceira execução do mesmo programa):
Resultado: Primeiro 1 Segundo 2 Terceiro 3
Indice 0: Primeiro
Indice 1: 1
Indice 2: Segundo
Indice 3: 2
Indice 4: Terceiro
Indice 5: 3

5. Conversão %hash = $escalar
Quando uma variável denotada por %hash recebe a atribuição de um valor escalar escrito na forma $escalar, o Perl interpreta o valor escalar como único elemento da hash, sendo este uma chave. Exemplo:

```
#Exemplo 1
%hash = (
```

```
    'Primeiro' => 1,
    'Segundo' => 2,
    'Terceiro' => 3
);
$escalar = 4;
%hash = $escalar;

foreach $key (keys %hash) {
    print "Chave: $key\n";
}
foreach $value (values %hash) {
    print "Valor: $value\n";
}
```

Output 1:
Chave: 4
Valor:

```
#Exemplo 2
%hash = (
    'Primeiro' => 1,
    'Segundo' => 2,
    'Terceiro' => 3
);
$escalar = “Quatro”;
%hash = $escalar;

foreach $key (keys %hash) {
    print "Chave: $key\n";
}
foreach $value (values %hash) {
    print "Valor: $value\n";
}
```

Output 2:
Chave: Quatro
Valor:

6. Conversão %hash = @array
Quando uma variável tipo hash recebe um array em uma atribuição, o Perl entende que cada par de posições do array corresponde a um par chave valor da hash. Mas novamente não há garantia de ordem na alocação dos pares na hash. Exemplo:

```
@array = ("primeiro", 1, "segundo", 2, "terceiro", 3);
%hash = @array;

foreach $key (keys %hash) {
    print "$key => $hash{$key}\n";
```

```
}
```

Output 1 (primeira execução do programa):
primeiro => 1
terceiro => 3
segundo => 2

Output 2 (segunda execução do mesmo programa):
primeiro => 1
segundo => 2
terceiro => 3

Output 3 (terceira execução do mesmo programa):
terceiro => 3
primeiro => 1
segundo => 2

Tarefa 2.1

Escolha um programa escrito em uma linguagem estática e dê dois exemplos de nome/entidade para cada tempo de amarração
O programa em C abaixo, busca o maior valor contido um vetor C de tamanho 3, onde cada um dos seus elementos corresponde a soma dos elementos de mesmo índice nos vetores A e B, também de tamanho 3.

```
#include <stdio.h>
#define TAM 3

int main() {
   int vetA[TAM], vetB[TAM], vetC[TAM];
   int maior = 0;

   for (int i = 0; i < TAM; i++) {
      printf("Entre com numero do Vetor A: ");
      scanf("%d", &vetA[i]);
      printf("Entre com numero do Vetor B: ");
      scanf("%d", &vetB[i]);

      vetC[i] = vetA[i] + vetB[i];

      if (vetC[i] > maior)
        maior = vetC[i];
   }

   printf("Vetor C: [%d, %d, %d]\n", vetC[0], vetC[1], vetC[2]);
   printf("O maior Elemento do Vetor C é: %d\n", maior);
```

```
    return 0;
}
```

tempo de design: amarração dos nomes “if” e “for” como palavras reservadas.
tempo de implementação: tamanho de “int” e o tamanho de “%d”.
tempo de pré-processamento: valor da constante “TAM”.
tempo de compilação: semântica do símbolos “+”, “<” e “>” em seus contextos .
tempo de link-edição: implementação do “scanf” e “printf”.
tempo de carregamento: endereços das variáveis “i=0” e “maior=0” além do espaço de endereçamento dos vetores “vetA”, “vetB” e “vetC”.
tempo de execução: endereços das entradas escaneadas em “&vetA[i]” e “&vetB[i]” e o valor de “vetC[i]”.

Tarefa 2.2

Escolha um programa escrito em uma linguagem dinâmica e dê dois exemplos de nome/entidade para cada tempo de amarração
O programa em Python 2.7, abaixo, calcula a soma dos termos de uma Progressão Aritmética. A Quantidade de termos, a razão e o primeiro termo da PA, são fornecidos pelo usuário.

```
n = input("Quantidade de termos da PA:")
a1 = input("Primeiro termo da PA:")
r = input("Razão da PA:")
aux = n+1
s = 0
for i in range(1, aux):
    t = a1 + (i - 1) * r
    s = s + t
    print t
print "Soma dos termos:", s
```

tempo de design da linguagem: os statements “for” e “print” como palavras reservadas da linguagem.
tempo de compilação: o valor inicial da variável s.
tempo de execução: o valor das variáveis “n”, “a1” e “r” e além do endereço de “aux”, “t” e “s”.

Tarefa 3.1

Pegue um programa seu razoavelmente grande…
enumere todos os usos de memória estática, pilha e heap
o programa deve ter pelo menos 5 usos da pilha e da heap

O Programa em C abaixo efetua a soma de matrizes com alocação dinâmica, cujas entradas são lidas a partir de um arquivo.

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

int **MatAlloc(int linha, int coluna, FILE *fp);
int **SomaMatriz(int **mat1, int **mat2, int linha, int coluna);
void PrintMat(int **matriz, int linha, int coluna);


int main()
{
    FILE *fp;
    int l,c;
    int **A,**B,**C;
    char name[81];

    fgets(name,80,stdin);
    name[strlen(name)-1]='\0';
    fp=fopen(name,"r");
    if(fp==NULL){
        printf("ERRO AO ABRIR O ARQUIVO!");
        return 0;}
    fscanf(fp,"%d;%d",&l,&c);

    A=MatAlloc(l,c,fp);
    B=MatAlloc(l,c,fp);
    C=SomaMatriz(A,B,l,c);

    printf("%d;%d\n",l,c);
    PrintMat(C,l,c);

    fclose(fp);

    free(A);
    free(B);
    free(C);

return 0;
}


int **MatAlloc(int linha, int coluna, FILE *fp)
{
    int **matriz;
    int i,j;

    matriz=(int**)malloc(linha*sizeof(int*));
```

```
48      for(i=0;i<linha;i++){
49          matriz[i]=(int*)malloc(coluna*sizeof(int));
50          for(j=0;j<coluna;j++){
51              fscanf(fp,"%i",&matriz[i][j]);}
52      }
53  return matriz;
54  }
55
56
57  int **SomaMatriz(int **mat1, int **mat2, int linha, int coluna)
58  {
59      int **mat3;
60      int i,j;
61
62      mat3=(int**)malloc(linha*sizeof(int*));
63      for (i=0;i<linha;i++){
64          mat3[i]=(int*)malloc(coluna*sizeof(int));
65          for (j=0;j<coluna;j++){
66              mat3[i][j]=mat1[i][j]+mat2[i][j];}
67      }
68  return mat3;
69  }
70
71
72  void PrintMat(int **matriz, int linha, int coluna)
73  {
74      int i,j;
75      for(i=0;i<linha;i++){
76          for(j=0;j<coluna;j++){
77              fprintf(stdout,"%i",matriz[i][j]);
78              if(j!=coluna-1){
79                  putchar(' ');}
80          }
81          putchar('\n');
82      }
83  }
```

- Usos da memória estática: nenhum.

- Usos da pilha:
1) no ponteiro “FILE *fp” declarado localmente - linha 12 do programa.
2) na declaração da variável local “int l” - linha 13 do programa.
3) na declaração da variável local “int c” - linha 13 do programa.
4) no ponteiro “int **A” declarado localmente - linha 14 do programa.
5) no ponteiro “int **B” declarado localmente - linha 14 do programa.
6) no ponteiro “int **C” declarado localmente - linha 14 do programa.
7) na declaração local do vetor “char name[81]” - linha 15 do programa.

8) na declaração local da variável “int linha” que serve como parâmetro da função - linha 42 do programa.
9) na declaração local da variável “int coluna” que serve como parâmetro da função - linha 42 do programa.
10) no ponteiro “FILE *fp” declarado localmente que serve como parâmetro da função - linha 42 do programa.
11) no ponteiro “int **matriz” declarado localmente - linha 44 do programa.
12) na declaração da variável local “int i” - linha 45 do programa.
13) na declaração da variável local “int j” - linha 45 do programa.
14) no ponteiro “int **mat1” declarado localmente que serve como parâmetro da função - linha 57 do programa.
15) no ponteiro “int **mat2” declarado localmente que serve como parâmetro da função - linha 57 do programa.
16) na declaração da variável local “int linha” que serve como parâmetro da função - linha 57 do programa.
17) na declaração da variável local “int coluna” que serve como parâmetro da função - linha 57 do programa.
18) no ponteiro “int **mat3” declarado localmente que serve como parâmetro da função - linha 59 do programa.
19) na declaração da variável local “int i” - linha 60 do programa.
20) na declaração da variável local “int j” - linha 60 do programa.
21) no ponteiro “int **matriz” declarado localmente que serve como parâmetro da função - linha 72 do programa.
22) na declaração da variável local “int linha” que serve como parâmetro da função - linha 72 do programa.
23) na declaração da variável local “int coluna” que serve como parâmetro da função - linha 72 do programa.
24) na declaração da variável local “int i” - linha 74 do programa.
25) na declaração da variável local “int j” - linha 74 do programa.

- Usos da heap:
1) Na linha 25 do programa na chamada com a chamada da função “MatAlloc” a heap é usada duas vezes na função, pelos comando “(int**)malloc(linha*sizeof(int*))” e “(int*)malloc(coluna*sizeof(int))” escritos respectivamente nas linhas 47 e 49 do programa.
2) Na linha 26 do programa na chamada com a chamada da função “MatAlloc” a heap é usada duas vezes na função, pelos comando “(int**)malloc(linha*sizeof(int*))” e “(int*)malloc(coluna*sizeof(int))” escritos respectivamente nas linhas 47 e 49 do programa.
3) Na linha 26 do programa na chamada com a chamada da função “SomaMatriz” a heap é usada duas vezes na função, pelos comando “(int**)malloc(linha*sizeof(int*))” e “(int*)malloc(coluna*sizeof(int))” escritos respectivamente nas linhas 62 e 64 do programa.
4) Na linha 34 do programa a heap é utilizada no comando “free(A)”.
5) Na linha 35 do programa a heap é utilizada no comando “free(B)”.
6) Na linha 36 do programa a heap é utilizada no comando “free(C)”.

Tarefa 3.2

Pesquise sobre a pilha em alguma linguagem ou arquitetura…
Qual é o tamanho da pilha em bytes?
Quantas chamadas recursivas são suportadas?
Quantas locais eu posso guardar em cada chamada?
Cite suas fontes.

Em Java o tamanho da pilha na varia entre 320kB e 1024kB, sendo 320kB o limite padrão e 1024kB (1MB) o limite máximo para a JVM. O tamanho de pilha pode ser configurado pelo programador, respeitando tais limites. Considerando um tamanho de pilha de 320k, o número máximo de chamadas em funções recursivas é de 10473, já para o limite máximo de 1MB na pilha, são suportadas cerca de 7000 chamadas recursivas.

Referências:

1. Configuring Stack Sizes in the JVM:
https://www.baeldung.com/jvm-configure-stack-sizes

2. Stack Safe Recursion in Java - Manning:
https://freecontent.manning.com/stack-safe-recursion-in-java/#:~:text=Default%20stack%20size%20varies%20between,case%20is%20recursive%20method%20calls.

2. How To Set Stack Size to overcome java.lang.StackOverflowError:
https://blogs.oracle.com/saas-fusion-app-performance/how-to-set-stack-size-to-overcome-javalangstackoverflowerror

3. Find limit of recursion - Java:
https://rosettacode.org/wiki/Find_limit_of_recursion#Java

4. What is the maximum depth of the java call stack?
https://stackoverflow.com/questions/4734108/what-is-the-maximum-depth-of-the-java-call-stack

5. JVM stack size specifications:
https://stackoverflow.com/questions/37026018/jvm-stack-size-specifications

Tarefa 4.1

Qual é o máximo de memória em bytes que o programa a seguir usa em um determinado momento?
- Explique como você fez o cálculo.
- Cite as suas fontes.

```
int fat (int n) {
  if (n == 0) {
    return 1;
```

```
  } else {
    return n * fat(n-1);
  }
}

int vec[100];

void main (void) {
  for (int i=0; i<100; i++) {
    vec[i] = fat(i);
  }
}
```

O valor máximo de memória consumido pelo programa em determinado instante da execução é 800 Bytes.

Esse valor foi calculado da seguinte forma: existe um vetor declarado como inteiro de 100 posições. Cada inteiro ocupa um espaço de 4 bytes na memória. Ao executar o for cada posição do vetor é preenchida com o fatorial do índice “i”, também declarado como inteiro. Na primeira iteração, “i” vale zero e portanto o valor de n será empilhado e a função "fat" retorna o valor 1 que é armazenado no vetor, sendo então desempilhada a “variável n”. Na segunda iteração, quando i = 1, o valor de n = 1 será empilhado e devido a chamada recursiva, o valor de n = 0 também irá para pilha. Esse processo se repete durante toda execução do comando for, logo, quando i = 99, todas as posições de 4 bytes (um inteiro) do vetor já foram preenchidas, exceto a última e além disso existirão 100 valores na pilha devido a recursão: 99 chamadas pendentes (empilhadas) + o caso base da recursividade (n = 0). Portanto, temos os seguinte cenário:

100 posições de memória ocupadas na pilha pela recursão.
1 posição de memória ocupada na pilha pela variável local “i”.
99 posições do vetor ocupadas na região estática da memória: isso acontece pois durante a execução da chamada recursiva, antes do retorno da função, a última posição do vetor ainda não foi preenchida e, portanto, a pilha não foi liberada. É aqui que o programa atinge o pico de alocação de memória durante a execução.

Como em todos os casos a variável é do tipo int, contendo 4 Bytes, é possível calcular a seguinte expressão:

(100*4) + (1*4) + (99*4) = 800 Bytes.

Obs: Na resposta está exposto o raciocínio realizado para o cálculo e não houve pesquisa externa, por isso não está inclusa a citação das fontes nesse caso.

Tarefa 4.2

Implemente a função fat a seguir sem usar variáveis locais. Dica: você vai continuar precisando de uma pilha.

```
int fat (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fat(n-1);
  }
}

int vec[100];

void main (void) {
  for (int i=0; i<100; i++) {
    vec[i] = fat(i);
  }
}
```

Tarefa 4.3

Quais são as regiões de memória do Java e como elas são utilizadas pela JVM? Cite suas fontes.

Internamente, a JVM mantém uma área da memória dividida em duas partes: heap e pilha. O espaço de heap em Java é usado para alocação de memória dinâmica para objetos Java e classes, ou seja, nessa região são armazenados todos os objetos criados durante a execução da aplicação, com seus métodos carregados e metadados das classes. Esta parte da memória é dividida em gerações para organizar os objetos, de acordo com o tempo em que possuem referências válidas, motivo pelo qual a JVM consegue gerenciar a alocação de memória usando o mecanismo de coleta de lixo (Garbage Collector). A heap é então dividida em três gerações: a “Young”, onde novos objetos criados são armazenados; a geração “Old” ou “Tenured”, onde objetos mais antigos são armazenados. Normalmente o coletor de lixo move objetos da geração “Young” para a “Old” após certo tempo em que determinado objeto está “vivo. E por fim, a geração “Perm”, onde a JVM armazena metadados de classes e métodos. Já a pilha contém valores primitivos que são específicos para um método e referências a objetos que estão instanciados na heap. A pilha é dividida em quadros (frames) para cada método cujos dados não são compartilhados. Quando o método termina a execução, o quadro de pilha correspondente é liberado, o fluxo retorna ao método de chamada e o espaço fica disponível para o próximo método. A memória de pilha cresce e diminui à medida que novos métodos são chamados e retornados, respectivamente, alocando e desalocando automaticamente a região. Se essa memória estiver cheia, o Java lançará “java.lang.StackOverFlowError”. O acesso a essa região da memória é rápido quando comparado à memória heap. Além disso, essa memória é segura para threads, pois na JVM, cada thread opera em sua própria pilha. Em resumo, os novos

objetos em Java sempre são criados no espaço de heap e as referências a esses objetos são armazenadas na memória da pilha enquanto estiverem em execução.

Referências:

1. Arquitetura da JVM – Runtime Data Area – Heap:
http://www.mauda.com.br/?p=964

2. Uma breve introdução ao gerenciamento de memória em Java:
https://www.infoq.com/br/articles/intro-memoria-JVM/

3. Gerência de Memória em Java:
https://pt.slideshare.net/helderdarocha/gerncia-de-memria-em-java-parte-ii

4. O que são e onde estão a “stack” e “heap”?
https://pt.stackoverflow.com/questions/3797/o-que-s%C3%A3o-e-onde-est%C3%A3o-a-stack-e-heap

5. Pilha de memória e espaço de heap em Java:
https://www.codeflow.site/pt/article/java-stack-heap

Tarefa 5.1

Sobre o conceito de hiding...
Procure uma linguagem que não permita hiding. O que acontece quando o programador tenta esconder uma variável?
COBOL é um exemplo de linguagem que não permite hiding ou shadowing de variáveis. As primeiras versões da linguagem não possuíam diferenciação de escopo em declarações de variáveis sendo elas consideradas variáveis globais em todo o contexto do programa. Quando um programador COBOL tenta ocultar ou sombrear determinada variável, o compilador lança uma exceção acusando que o nome já está em uso, pois a linguagem não permite uso repetido de nomes nas declarações de variáveis.

Quais são as vantagens e desvantagens de permitir ou não permitir hiding?
O problema com uso do sombreamento é que pode ser difícil de o programador controlar: imagine uma função com dezenas de linhas e vários blocos aninhados e sequenciais. Se a função for longa o suficiente de forma que o programador não consiga ver todas as diferentes definições em diferentes escopos, é provável que ele cometa uma interpretação incorreta. Em outras palavras, o sombreamento de variáveis locais pode levar a erros inadvertidos do programador onde a variável errada é usada ou modificada. A vantagem de linguagens que possibilitam o hiding de nomes está associada, por exemplo a herança de classes em orientação a objetos. Dessa forma, uma classe filha pode ocultar as variáveis da classe pai, isso significa que as variáveis atributos da superclasse (pai) só estarão acessíveis utilizando o construtor da classe pai (super.variavel).

Referências:

1. Variable shadowing:
https://en.wikipedia.org/wiki/Variable_shadowing

2. Shadowed Classes or Variables in Java:
https://www.dummies.com/programming/java/shadowed-classes-or-variables-in-java/

3. How bad is redefining/shadowing a local variable?
https://stackoverflow.com/questions/38533407/how-bad-is-redefining-shadowing-a-local-variable

4. Global variables:
https://en.wikipedia.org/wiki/Global_variable

5. Por que as linguagens de programação permitem ocultar / ocultar variáveis e funções?
https://qastack.com.br/software/215091/why-do-programming-languages-allow-shadowing-hiding-of-variables-and-functions

6. What is the scope of COBOL:
https://stackoverflow.com/questions/5070102/what-is-the-scope-of-cobol

7. Variable Shadowing and Hiding in Java
https://dzone.com/articles/variable-shadowing-and-hiding-in-java

8. Rule: no-shadowed-variable:
https://palantir.github.io/tslint/rules/no-shadowed-variable/#:~:text=Rationale,value%20an%20identifier%20actually%20refers.

Tarefa 5.2

Sobre escopo dinâmico…
Quais outras linguagens possuem escopo dinâmico? Como o escopo dinâmico funciona nela?
Dê um exemplo interessante do seu uso.
Além do Perl, outros exemplos de linguagens que utilizam escopo dinâmico são: Common LISP, Emacs LISP, APL, Logo, Bash, Power Shell e Shell Script.

Como funciona o escopo dinâmico em Shell Script:
No Shell Script, um nome é resolvido a partir do contexto de execução, isso significa que o runtime do shell pesquisa primeiro uma amarração da variável na função local, depois pesquisa na função que chamou a função local, em seguida na função que chamou essa função e assim por diante, progredindo na pilha de chamadas de função.

Exemplo:

```
dir="/home"

remove_image(){
   rm $dir/*.jpg;
}

clear_current_dir(){
   local dir=$(pwd);
   remove_image;
}

clear_current_dir
cd $dir
```

Tarefa 5.3

Sobre o tratamento de exceções em Java ou Python…
A partir de um erro (throw/raise), como determinar o ponto de captura da exceção (catch/except)?
Em Python, um erro é capturado a partir do contexto onde a exceção aconteceu, na forma de um rastreamento de pilha. Em geral, um rastreamento de pilha lista as linhas do programa contendo as chamadas dos módulos onde o erro se originou. No Python, o módulo traceback é responsável por rastrear essas exceções a partir de um ponto específico do código. Esse módulo trabalha com a pilha de chamadas para produzir mensagens de erro. Isso significa que dado um statement “raise” como manipulador de uma exceção, o traceback realiza um rastreamento na pilha - a partir do ponto da declaração “raise” - seguindo de baixo para cima na cadeia de chamadas das funções, da mais recente para a menos recente, até encontrar o ponto onde a exceção foi gerada. Ao finalizar a pesquisa pela exceção o traceback exibe a sequência de chamadas representadas por entradas de duas linhas para cada chamada. A primeira linha de cada chamada contém informações como nome do arquivo, número da linha e nome do módulo, todos especificando onde o código pode ser encontrado. A segunda linha para essas chamadas contém o código real que foi executado.

Tratamento de exceções é mais parecido com escopo estático ou dinâmico? Justifique.
Os tratamentos de exceção tem mais similaridade com escopo dinâmico pois avaliam o contexto de execução e suas amarrações. Isso é possível devido a capacidade de empilhar as chamadas de função que garante a rastreabilidade dos erros a partir de determinado ponto de execução no programa.

Módulo 3 - Programação Funcional

Tarefa 1.1

Sobre o conceito de "transparência referencial"...
1. Explique esse conceito em uma frase com as suas próprias palavras.
Dado que para as mesmas entradas determinada expressão produz as mesmas saídas, a transparência referencial traduz a capacidade que essa expressão possui de inalterar um programa ao ser substituída por seu valor correspondente, isto é, pela sua própria saída.

2. Quais são as principais vantagens de transparência referencial?
A transparência referencial torna o código mais íntegro, menos suscetível a erros e mais conciso, de forma que as funções podem ser chamadas várias vezes, sabendo que não ocorrerão efeitos colaterais. Isso permite que o programador tenha maior controle intelectual sobre o programa escrito. A transparência referencial também torna cada subprograma independente, o que simplifica muito o teste unitário e a refatoração. Como um benefício adicional, programas referencialmente transparentes são mais fáceis de ler e entender.

3. Por quê essas vantagens são possíveis?
Essas vantagens são possíveis por dois motivos principais: o primeiro é a forma como as variáveis se comportam na programação funcional, mantendo os mesmos valores declarados até o fim da execução do programa. O segundo está relacionado ao fato de as funções serem puras e determinísticas, isto é, retornam sempre o mesmo resultado para para as mesmas entradas.

Tarefa 1.2

Sobre funções recursivas…
1. Dê exemplos de funções recursivas que você já tenha escrito em trabalhos e projetos. (Faça uma busca rigorosa.)
O código abaixo foi implementado em um exercício da disciplina de Linguagem de Programação 1 (LP1) e trata-se de um algoritmo recursivo em C que imprime conteúdo de arquivo de trás para frente.

```
#include <stdio.h>
#include <string.h>

void le_arq(char [], char []);
void inverteRec(char v[], int i);

int main()
{
        char nome[81];
        char vet[800];
```

```
	fgets(nome, 80, stdin);
	nome[strlen(nome)-1]='\0';
	le_arq(vet,nome);
	inverteRec(vet, 0);
}

void le_arq(char v[], char nome[])
{
	FILE * fp = fopen(nome, "r");
	int i=0;
	char c;

	while ((c=fgetc(fp))!=EOF){
		v[i]=c;
		i++;}
	v[i]='\0';
	fclose(fp);
}

void inverteRec(char v[], int i)
{
	if (v[i]!='\0'){
		inverteRec(v, i+1);
		printf("%c", v[i]);}
}
```

2. Por quê você optou por essa técnica?
A recursão nesse caso foi uma boa alternativa pois elimina a necessidade de declarar e percorrer um novo vetor auxiliar (além do utilizado na captura dos caracteres) como estrutura de armazenamento para garantir que o conteúdo seja invertido, economizando a alocação na memória estática.

Tarefa 1.3

Sobre funções de alta ordem…
1. Dê exemplos de funções de alta ordem que você já tenha escrito ou usado em trabalhos e projetos. (Faça uma busca rigorosa.)
2. Por quê você optou por essa técnica?

Tarefa 1.4

O exemplo a seguir não adere ao modelo funcional pois possui duas declaração para a variável b. Mas o que deveria acontecer quando o programador comete esse erro?

a = b * 2

x = a + b
b = 10
b = 0

Considerando uma abordagem onde a ordem de avaliação é explícita, uma exceção seria lançada ao tentar executar o programa, por falta de declaração da variável “b”. Contudo, considerando um contexto, embora não funcional, mas onde as a avaliação não é realizada explicitamente de forma ordenada, a execução do programa apresentado poderia levar a resultados ambíguos, onde ora “b” vale 0, ora vale 10, portanto, nesse cenário “x” poderia resultar em 0 ou 30. Por fim, considerando que, apesar de o exemplo não aderir ao modelo funcional, em paradigma funcional as variáveis possuem valores imutáveis até o fim da execução, o programa apresentado também geraria um erro e não seria executado pois a variável “b” não pode assumir dois valores, uma vez que não existe mudança de estado nesse paradigma.

Tarefa 3.1

Execute o passo-a-passo de ambas para as seguintes chamadas:
foldr (-) 0 [1,2,3,4]
foldl (-) 0 [1,2,3,4]
Essas funções são equivalentes? Por quê?

Execução da função foldr (-) 0 [1,2,3,4]:
[1,2,3,4] 0
[1,2,3,4]
[1,2,1]
[1,-1]
-2

Execução da função foldl (-) 0 [1,2,3,4]:
0 [1,2,3,4]
  [-1,2,3,4]
    [-3,3,4]
      [-6,4]
        -10

As funções não são equivalentes. O resultado será equivalente apenas para operações comutativas, como no caso da soma ou multiplicação. Outras operações como subtração, divisão, cálculo de resto e até mesmo operações não numéricas como a concatenação, resultarão em retornos distintos ao utilizar foldr ou foldl.

Tarefa 3.2

a. Dê 2 exemplos de usos reais para a função map.

b. Dê 2 exemplos de usos reais para a função filter.
c. Dê 2 exemplos de usos reais para a função fold.

Os exemplos precisam ser concretos com a descrição de um cenário real (ex., financeiro, jogos, estatística, etc).
Não é necessário implementá-los por completo, basta a chamada às funções em questão.
Não use exemplos de operações que já foram vistas nos vídeos (ex. fold (+) ...).
A partir de agora, use Haskell em todas as respostas.

a) Exemplos reais da função map:

1. Pode, por exemplo, ser usada em aplicações cujo objetivo é identificar as palavras que mais aparecem em postagens de uma rede social, removendo elementos indesejados.

-- a função “splitOn” usa o separador espaço para segmentar a string e alocar cada parte separada em uma posição da lista. O resultado será: [“Your”, “words”, “are”, “very”, “valuable”, “on”, “social”, “media,”, “believe”, “me!”]
splitedtext = splitOn " " "Your words are very valuable on social media, believe me!"

-- aplica uma função “remove_punctuation” para remover a pontuação em cada elemento da lista “splitedtext”.
words = map remove_punctuation splitedtext

-- O resultado será a nova lista: [“Your”, “words”, “are”, “very”, “valuable”, “on”, “social”, “media”, “believe”, “me”]
main = print words

2. Pode, por exemplo, ser utilizada em uma aplicação que estima o valor futuro de investimentos em títulos (juros compostos), com rendimento de 6% ao ano, para os próximos 5 anos.

-- função que faz o cálculo dos juros, considerando o percentual de rendimento e o tempo do investimento.
juros_compostos n = n * (1 + 0,06) ^ 5

-- lista que contém os valores iniciais aportados.
aportes = [1000, 500, 250, 400, 100]

-- aplica a função “juros_compostos” em cada elemento da lista “aportes”.
valor_futuro = map juros_compostos aportes

-- o resultado será a nova lista: [1338.23, 669.11, 334.56, 535.29, 133.82]
main = print valor_futuro

b) Exemplos reais da função filter:

1. Pode, por exemplo, ser utilizada em uma aplicação cuja a finalidade é definir as placas que farão parte do rodízio de veículos no estado de SP, a partir da letra inicial da placa do automóvel.

```
-- lista encadeada contendo as placas elegíveis.
placas = ["LVK-4135", "HRQ-4184", "LQW-2346", "LVF-2818", "KQJ-5279"]

-- filtra a lista de placas, por meio da função anônima passada como argumento, cuja finalidade é identificar as cadeias que inicial pela letra “L”.
rodizio = filter (\x -> take 1 x == "L") placas

-- o resultado será a nova lista: ["LVK-4135","LQW-2346","LVF-2818"]
main = print rodizio
```

2. Pode, por exemplo, ser utilizada em uma aplicação para identificar suspeitas de casos de covid-19 alertando ocorrência de febre, isto é temperaturas corporais acima de 37.5 °C.

```
-- lista encadeada de temperaturas aferidas em pacientes suspeitos.
temp = [35.5, 38.9, 37.1, 36.4, 35.8, 36.3, 36.9]

-- filtra a lista através da função anônima passada como parâmetro, que identifica temperaturas superiores a 37.5.
febre = filter (\x -> x > 37.5) temp

-- o resultado será a nova lista: [38.9]
main = print febre
```

c) Exemplos reais da função folder:

1. Pode, por exemplo, ser utilizada em uma aplicação que contabiliza o tempo total de permanência de um usuário em um endereço web (em milissegundos), de acordo com o tempo em cada página do mesmo link.

```
-- lista encadeada contendo o tempo de permanência em ms, por página no mesmo endereço.
tempo_por_pagina = [150000, 102340, 69876, 59871]

-- acumula os valores de tempo presentes na lista passada como argumento.
tempo_total = foldr (+) 0 tempo_por_pagina

-- o resultado será: 382087
main = print tempo_total
```

2. Pode, por exemplo, ser utilizada em uma aplicação cuja a finalidade é gerar um token para uma credencial de acesso, dada uma lista com elementos alfanuméricos gerados de forma aleatória.

```
 -- lista encadeada de elementos alfanumericos gerados aleatoriamente.
alphanum = ["7", "9", "a", "3", "b", "k", "g", "h", "1", "2", "3", "6"]
-- concatenação dos elementos alfanuméricos
token = foldl (++) "" alphanum
-- o resultado será: "79a3bkgh1236"
main = print token
```

Módulo 4 - Tipos Paramétricos

Tarefa 1.1

Considere um jogo estilo RPG com Guerreiros, Magos e Sacerdotes...
1. Crie um tipo de dados em Haskell que represente as três classes acima. Considere que algumas propriedades são comuns às três classes (ex., altura e idade) e outras não (ex., sacerdotes rezam).
2. Crie o mesmo tipo de dados em C usando enum, struct, union. Enumere e explique as propriedades das classes antes de criar os tipos.

1) Guerreiro:
Características físicas: peso, altura, força; // Máx. força = 100
Características pessoais: idade, sexo, hp, xp; // Sexo = m ou f e Máx hp = Máx xp = 5000
Habilidades especiais: lutar, cavalgar; // Máx = 5000 para todas as habilidades especiais

2) Mago:
Características físicas: peso, altura, força; // Máx. força = 100
Características pessoais: idade, sexo, hp, xp; // Sexo = m ou f e Máx hp = Máx xp = 5000
Habilidades especiais: magia, autodefesa; // Máx = 5000 para todas as habilidades especiais

3) Sacerdote:
Características físicas: peso, altura, força; // Máx. força = 100
Características pessoais: idade, sexo, hp, xp; // Sexo = m ou f e Máx hp = Máx xp = 5000
Habilidades especiais: rezar, curar; // Máx = 5000 para todas as habilidades especiais

1. IMPLEMENTAÇÃO EM HASKELL

```
data Personagem =   Guerreiro Float Float Int Int Char Int Int Int Int
                  | Mago Float Float Int Int Char Int Int Int Int
                  | Sacerdote Float Float Int Int Char Int Int Int Int
p :: Personagem
p = Guerreiro 81 1.85 100 35 m  2000 3000 4000 2500
```

2. IMPLEMENTAÇÃO EM C

```
enum PERSONAGEM { GUERREIRO, MAGO, SACERDOTE }
struct Personagem {
   enum PERSONAGEM classe;
   union {
      struct { float peso; float altura; unsigned int forca;
               unsigned int idade; unsigned char sexo; unsigned int hp; unsigned int xp;
               unsigned int lutar; unsigned int cavalgar;       }; // GUERREIRO
      struct { float peso; float altura; unsigned int forca;
               unsigned int idade; unsigned char sexo; unsigned int hp; unsigned int xp;
               unsigned int magia; unsigned int autodefesa; }; // MAGO
      struct { float peso; float altura; unsigned int forca;
               unsigned int idade; unsigned char sexo; unsigned int hp; unsigned int xp;
               unsigned int rezar; unsigned int curar;          }; // SACERDOTE
   };
};
struct Personagem p = { MAGO { 68, 1.75, 30, 90, f, 4000, 4800, 5000, 3000  } };
```

Tarefa 2.1

Considere a definição paramétrica de listas a seguir…
data Lista a = No a (Lista a) | Vazio
Crie uma lista que guarde sublistas de inteiros.

```
lista_1 :: Lista Int
lista_1 = No 1 (No 2 (No 3 Vazio))

lista_2 :: Lista Int
lista_2 = No 4 (No 5 (No 6 Vazio))

lista_3 :: Lista Int
lista_3 = No 7 (No 8 (No 9 Vazio))

lista :: Lista Lista
lista = No lista_1 (No lista_2 (No lista_3 Vazio))
```

Tarefa 2.2

1. Crie uma árvore binária de booleanos:

```
data Arvore a  =  Galho (Arvore a) a (Arvore a)
                | Folha

arvorebin :: Arvore Bool
```

```
arvorebin = Galho (Galho (Galho Folha true Folha)
                         false
                        (Galho Folha true Folha))
                  false
                 (Galho (Galho Folha true Folha)
                         false
                        (Galho Folha true Folha))
```

2. Crie uma árvore em que cada nó guarda uma lista de inteiros:

```
data Lista a = No a (Lista a) | Vazio

lista_1 :: Lista Int
lista_1 = No 1 (No 2 (No 3 Vazio))

lista_2 :: Lista Int
lista_2 = No 4 (No 5 (No 6 Vazio))

lista_3 :: Lista Int
lista_3 = No 7 (No 8 (No 9 Vazio))

data Arvore a  =  Galho (Arvore a) a (Arvore a)
               | Folha

arvorebin :: Arvore Lista Int

arvorebin = Galho (Galho (Galho Folha lista_1 Folha)
                         lista_2
                         (Galho Folha lista_3 Folha))
```