Lista de Exercícios de Linguagens de Programação I
Universidade Federal do Amazonas/Instituto de Computação

Alunos: Bruno Pinheiro - 21650919
Priscila Barros - 21201437

Introdução

1) Qual o principal objetivo da criação do Fortran, que contrapunha linguagens de primeira e segunda geração? Especificamente, qual o público-alvo do Fortran?

Fortran foi desenvolvida por volta de 1954 e 1957, é considerada a primeira linguagem, por ter sido um revolução na época, foi a primeira linguagem imperativa, antes da linguagem Fortran todos com programas de computadores eram muito lentos e originaram muitos erros. Foi uma linguagem que se tornou bastante popular, diminuiu os erros de programação e possuía compilador que gerava código de qualidade, foi destinada a cientistas, engenheiros e analistas numéricos.

2) Embora muito pouco usada, o Algol teve enorme influência sobre projetos modernos. Cite características do Algol observadas em linguagens modernas.

O Algol foi a primeira linguagem de Programação estruturada,assim influenciando fortemente as linguagens mais novas, a especificação da linguagem também criou desafios técnicos para a construção de compiladores. O Algol possui as seguintes características: clareza na sua estrutura, baseada nos blocos e o estilo de sua definição, que usa uma linguagem metalingüística para definir de forma concisa e relativamente completa a sua sintaxe.

3) Uma meta de projeto comum de C e Java foi a portabilidade. Muitas das características adotadas por C para atingi-la levaram à criação de uma linguagem potencialmente insegura (para se ter uma linguagem pequena e simples, optou-se por pouco rigor em checagem de tipos, o controle de recursos como memória foi deixado para o programador, etc). Que solução foi adotada por Java para o mesmo objetivo sem sacrificar segurança? O que foi sacrificado neste caso?

Linguagens como o C utilizam um compilador diferente para cada SO, e assim o código pode ser facilmente portável pois poderá rodar em diversos sistemas operacionais sem muitas alterações, mas sempre será necessário recompilar o código gerando um binário compatível com a plataforma em questão, tendo como vantagem um melhor desempenho porque sempre teremos um binário otimizado para cada plataforma, no entanto acaba tornando a linguagem menos segura pois o binário torna-se vulnerável pois qualquer ataque externo terá acesso diretamente ao hardware. O Java é muito conhecido por ser uma linguagem multiplataforma, possibilitando que seus programas possam ser executados por qualquer sistema operacional, pois possui uma JVM(Java Virtual Machine) responsável por converter o código Java em comandos que o sistema operacional possa executar, este binário gerado pela JVM que pode ser executado em qualquer plataforma é chamado de bytecode, dando mais segurança para o binário escrito pois em um possível ataque, o mesmo não terá acesso direto ao hardware apenas a JVM, mas foi necessário tornar o programa mais pesado do que seria escrito se fosse feito diretamente na plataforma.

4) Qual a diferença entre linguagens de script e linguagens de finalidade geral?

O modo de traduzir o programa para binário e assim ser executado pelo sistema operacional diferencia as linguagens de script e de finalidade geral as linguagens de Script fazem uso de interpretadores que traduzem trechos seguidos de sua execução imediata, assim podemos dizer que o código foi interpretado, pois o código é executado simultaneamente a construção de outros trechos de código, linguagens como JavaScript e Python são exemplos de linguagens que utilizam script.
Já as linguagens de finalidade geral para converter seus códigos fontes fazem uso de métodos como compiladores ou máquinas virtuais.
O método de compilação diretamente pelo compilador traduz o código fonte para um código executável, fazendo uso de um compilador específico para determinado sistema operacional, sendo necessário todo o programa ser escrito completamente e salvo para assim compilar o mesmo e gerar a versão executável, a versão compilada é salva para que possa ser feita novas compilações, linguagens como C e Pascal são exemplos de linguagens que utilizam compiladores.
Normalmente os programas interpretados são mais flexíveis, no entanto também são mais lentos do que os compilados.
Há possibilidade de programas serem compilados para uma máquina virtual gerando um código de máquina que poderá ser executado por qualquer SO. JAVA e C# são exemplos de linguagens que utilizam esse processo.

5) Que linguagens **introduziram** os principais paradigmas de programação, em termos de **prevalência no tempo e impacto** sobre as linguagens futuras?

Tendo em vista que os principais paradigmas de programação são: o Imperativo, a orientação a objeto, o funcional , o lógico e o declarativo, podemos iniciar pelo **paradigma Imperativo:**

- Teve como primeira linguagem de alto nível com uma ampla aceitação o Fortran (FORmula TRANslation). Que foi iniciado para ações científicas e contou com uma notação algébrica, tipos, subprogramas, e entrada/saída formatada. Continua a ser largamente utilizado na comunidade de computação científica. No entanto outra linguagem que surgiu próxima ao FORTRAN e também teve um forte impacto sobre o paradigma imperativo foi o ALGOL 60 (ALGorithmic Oriented Language) dando origem ao B, ao ADA, ao Pascal e ajudou a dar origem ao Scheme (apesar de o scheme ser uma linguagem pertencente ao paradigma funcional), o ALGOL foi concebido em 1960 por um comitê internacional para uso em resolução de problemas científicos, ele introduziu a notação BNF para a definição de sintaxe e é um modelo de clareza e completude, também introduziu estrutura de bloco, instruções de controle estruturados e procedimentos recursivos no paradigma de programação imperativa.

Paradigma funcional:

- A primeira linguagem de programação funcional criada para computadores foi LISP, mesmo não sendo uma linguagem de programação puramente funcional, LISP introduziu a maioria das características hoje encontradas nas modernas linguagens de programação funcional e também foi a linguagem que deu origem ao Common Lisp, ao Logo e ao Scheme, que foi uma tentativa posterior de simplificar e melhorar LISP. Outra linguagem que também causou impacto nesse paradigma foi o ML criada pela Universidade de Edimburgo, pois posteriomente deu origem a linguagem Miranda na Universidade de Kent e que por sua vez deu origem ao Haskell que foi lançada no fim dos anos 1980 numa tentativa de juntar muitas ideias na pesquisa de programação funcional.

Paradigma orientado a objeto:

- A linguagem Simula 67 foi projetada para apoiar a Simulação de Eventos Discretos, criadas entre 1962 e 1968 por Kristen Nygaard e Ole-Johan Dahl no Centro Norueguês de Computação em Oslo. Esta linguagem gerou um enorme impacto pois foi a primeira linguagem orientada a objetos. No entanto foi necessário algum tempo para que esta linguagem pudesse se consolidar. Orientação a objeto teve que esperar a criação da Smalltalk para definir seu impacto, que foi uma linguagem gerada através da Simula 67, assim como Simula deu origem ao Eiffel e ajudou a dar origem ao C++.

Paradigma Declarativo:

- O Prolog foi a primeira linguagem declarativa e a que dispôs mais impacto neste paradigma, usa uma coleção base de dados de fatos e de relações lógicas (regras) que exprimem o domínio relacional do problema a resolver. Um programa pode rodar num modo interativo, a partir de consultas (queries) formuladas pelo usuário.O Prolog é baseado num subconjunto do cálculo de predicados de primeira ordem, o que é definido por cláusulas de Horn.

Paradigma Lógico:

- O Planner foi a primeira linguagem voltada para o paradigma lógico, possibilitando a orientação a padrões de planos procedimentais de asserções e de objetivos. No entanto tinha padrões de memória limitados, pois eram os padrões limitados da época. Posterior ao Planner veio a linguagem Prolog que por sua vez tinha o objetivo de simplificar o Planner, e que o mesmo deu origem ao QA-4, ao Popler, ao Conniver, e ao QLISP. Já o Prolog deu origem ao Mercury, ao Visual Prolog, ao Oz e ao Frill.

Aquecimento em Programação Funcional

6) Defina os seguintes conceitos como são compreendidos do paradigma funcional: (a) funções anônimas, (b) funções de primeira classe, (c) funções de alta ordem, (d) funções recursivas, (e) funções puras, (f) avaliação preguiçosa, (g) iteradores e (h) geradores ou streams

a. **Funções anônimas:**
   Funções que não possuem um identificador. Usadas normalmente quando é necessário passar uma função como parâmetro para uma função de alta ordem.
   Ex:

```
def ehPrimo(n):
      return len(list(filter(lambda k: n%k==0, range(2,n)))) == 0
```

b. **Funções de primeira classe**
   Uma linguagem suporta funções de primeira classe quando trata funções como "objetos de primeira classe" (como valores que são passados, manipulados, retornados, etc.) Isto significa que a linguagem passa funções como argumento para outras funções, as retorna como valor e variáveis podem recebê-la.

   Ex:

```
def caps(texto):
    return texto.upper()
```

```
print caps('ola mundo')
aux = caps
print aux('ola mundo')
```

c. Funções de alta ordem

Funções que aceitam outras funções como argumentos, são chamadas funções de alta ordem. Em python funções como .map() e .filter() são funções que estão nesta categoria, pois consomem funções como argumento.

Ex:

Quadrado dos números de 1 a 5

```
list(map(lambda x: x*x, range(1,6)))
>>>[1,4,9,16,25]
```

d. Funções recursivas

Função que que pode chamar/invocar a si mesma. A ideia consiste em definir um caso base, cujo resultado é conhecido de imediato, e um passo recursivo, no qual se tenta resolver um sub-problema do problema inicial, até atingir o caso base.

Ex: Fatorial de n

```
def fact(n):
    if n == 0:
      return 1
    else:
      return n * fact(n-1)
```

e. Funções puras

O conceito desta função é que ela pode ser substituída pelo seu valor de retorno(definição matemática). Neste contexto uma função f que recebe como argumento x, sempre resolve para y, independente do estado do sistema onde está incluída e em contrapartida não altera de forma transparente o sistema quando é aplicada

f. Avaliação preguiçosa

É uma técnica/avaliação/execução de procedimentos sob uma coleção. É utilizada para atrasar a computação sobre uma coleção, quando o procedimento pode não ser necessário(ponto em que o resultado é suficiente), poupando tempo de execução

Ex: função primo

```
primo = lambda n: True not in (n%x == 0 for x in range(2,n))
```

g. Iteradores

Sequência de elementos geradas por meio de avaliação preguiçosa. Normalmente implementados através de um método next() que retorna o próximo elemento da coleção.

h. Geradores ou *streams*

Geradores são tipos especiais de iteradores (funções ou expressões) que iteram sob coleções como também à geram dinamicamente.

7) Usando LCs, defina a função *pares(numero)* que retorna o número de dígitos pares em *numero*.

```
ehPar = lambda x: x%2==0

def pares(numero):
    soma = 0
    lista = str(numero)
    for i in range(0,len(lista)):
        if ehPar(int(lista[i])):
            soma = soma + 1
    print soma
```

8) Defina a função *menos_que_k_vogais(string)* que retorna as palavras em *string* com menos que *k* vogais.

```
def  ehVogal(l):
      vogais = ['a','e','i','o','u']
      return l.lower() in vogais

def conta_vogais(string):
      soma = 0
      for i in range(0,len(string)):
      if(ehVogal(string[i])):
            soma = soma+1
      return soma

def menos_que_k_vogais(string, k):
      lista = string.split(" ")
      resposta = []
      for i in range(0,len(lista)):
      if(conta_vogais(lista[i])==k):
            resposta.append(lista[i])
      print resposta
```

9) Implemente a função *maioria* que tem como parâmetros uma propriedade (descrita como uma função anônima) e uma lista e retorna True se mais que metade dos elementos na lista têm a propriedade.

```
def maioria(func, lista):
      cont = 0
      for i in range(0,len(lista)):
            if(func(lista[i])):
                  cont = cont + 1
      if(cont>(len(lista)/2)):
            return True
      else:
```

```
            return False
```

10) Implemente a função composição que tem como parâmetros uma lista de funções e um número e retorna o valor resultante da composição das funções aplicadas ao número.

```
def composicao(lista_de_funcoes, numero):
    var = numero
    for i in range(0,len(lista_de_funcoes)):
        funcao = lista_de_funcoes[i]
        var = (funcao(var))
    print var
```

O jogo de Bacará

12) Crie uma função que dada uma lista lst e um valor k, retorne lst sem o k-ésimo elemento.
Ex: sem_elemento_k([1,2,3],1) = [1,3]

```
def sem_elemento_k(lista, k):
    lista.remove(lista[k])
    return lista
```

13) Crie uma função que dada uma lista retorne uma versão embaralhada da mesma.
Ex: *deque_embaralhado([1,2,3,4,5]) = [2,3,5,4,1]* usando recursão e a função *sem_elemento_k*.

```
from random import randint
def deque_embaralhado(deque):
    lista = []
    for i in range(len(deque)):
        k = randint(0,len(deque)-1)
        lista.append(deque[k])
        sem_elemento_k(deque, k)
    return lista
```

14) Uma mão é uma sequência de cartas. Escreva uma função que, dada uma sequência de cartas na forma de uma lista, produza uma string correspondente à lista de cartas.

```
def lista_mao(lista):
    mao = []
    for i in range(len(lista)):
        mao.append(lista[i][0] + lista[i][1])
    return mao
```

15) Escreva uma função para avaliar uma mão de cartas no jogo bacará. Neste jogo, as cartas são avaliadas como segue. 10, Rei(K), Rainha(Q), Valete(J) valem 0 pontos cada. As cartas de 2 a 9

valem seus valores de face. A carta Às(A) vale 1 ponto O valor da mão corresponde à soma dos valores das cartas, descontada a dezena.

Foi necessário a criação da função *retorna_valor* devido a linguagem python não suportar nativamente um *"switch case"*

```
def retorna_valor(carta):
    if(carta[0][0]=='A'):
        return 1
    elif (carta[0][0]=='2'):
        return 2
    elif (carta[0][0]=='3'):
        return 3
    elif (carta[0][0]=='4'):
        return 4
    elif (carta[0][0]=='5'):
        return 5
    elif (carta[0][0]=='6'):
        return 6
    elif (carta[0][0]=='7'):
        return 7
    elif (carta[0][0]=='8'):
        return 8
    elif (carta[0][0]=='9'):
        return 9
    else:
        return 0

def valor_mao(lista):
    soma = 0
    for i in range(len(lista)):
        soma =  soma + retorna_valor(lista[i][0])
    return soma%10
```

16) Escreva uma função para avaliar uma mão de cartas no jogo bacará. Neste jogo, as cartas são

19) Baseado em sua experiência com Python Funcional, a compare com C, considerando os seguintes critérios:

a.Ortogonalidade, Tipos de Dados, Projeto de Sintaxe, Suporte à Abstração, Expressividade, Checagem de Tipos, Manipulação de Exceções e Restrições de Aliases.

	Python Funcional	C
Ortogonalidade	É uma linguagem mais sucinta, sem muitas exceções às regras	Baixa ortogonalidade, quase todas as combinações

		primitivas são aceitas o que confundem o compilador no que pode passar ou não .
Tipos de Dados	booleanos, numéricos(inteiros, ponto flutuante), lista, strings, tuplas	unsigneds, longs, int, float
Projeto de Sintaxe	Python preza pela identação do código, para se tornar uma linguagem altamente legível. Não usa chaves para delimitar blocos, nem termina sentenças utilizando ponto e vírgula.	Foi desenvolvida por programadores familiarizados com Assembly, logo sua sintaxe lembra em alguns momentos programas de baixo nível
Suporte à Abstração	Apenas no paradigma OO	Fraco suporte a abstração, para utilizar estruturas ou fazer operações complexas é necessário que a construção esteja com todos os detalhes
Expressividade	Não é uma linguagem muito “verbosa”, os algoritmos são mais sucintos e ter conhecimento de suas funções (.map() e lambda() e.g.) torna a linguagem muito expressiva	A linguagem permite escrever a mesma coisa de diversas maneiras (pouca ortogonalidade), podendo causar dúvidas sobre a funcionalidade do programa.
Checagem de Tipos	Suporta tipagem dinâmica (dados, variáveis) e forte (avaliação das expressões).	É linguagem de tipagem fraca, permite que um dado seja acessado como sendo de um tipo diferente do especificado.
Manipulação de Exceções	Possui um recurso de tratamento de exceções que retiram do programador a responsabilidade de tratamento	O programador é responsável por tratar as exceções manualmente
Restrições de Aliases	Diferentes nomes para o mesmo objeto, em exceto os tipos primitivos	É possível utilizando ponteiros

b. Legibilidade, escrita e confiabilidade.

	Python Funcional	C
Legibilidade	Possui simplicidade para ser lido, principalmente devido a sua indentação que permite que o visual do código seja bem visto e separado.	difícil legibilidade devido a sua falta de ortogonalidade, distinção de passagens de valor ou por referência, permitido o uso do goto.
escrita	possui simplicidade na escrita, pois consegue abstrair detalhes na estrutura, facilidade de leitura.	escrita não é tão fácil,várias possibilidades de escrever o mesmo pedaço de código, dificuldade na leitura de códigos extensos ou medianos.
confiabilidade	não é tão confiável, não possui tratamento de exceção, mas possui verificação de tipos	Não é tão confiável, verificação fraca de tipos, não possui tratamento de exceções.