UEPB Universidade Estadual da Paraíba

CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

PARADIGMAS DE PROGRAMAÇÃO

ATIVIDADE 3

Aluno: ***Luiz Carlos de Lemos Júnior*** – Matrícula: ***212080423***

1) Utilizando Prolog...

a) Escreva um código com os seguintes fatos/regras:

- Alexsandro, Antônio, Arton, Britney, Bruno, Caio, Camila, Fernando, Gabriel, Ian, Joanderson, Kelvyn, Luana, Lucas, Luiz, Maiara, Marcelo, MateusViana, MateusSilva, Mikaelle, Natália, Thiago, Victor e Wilton cursam PP;
- Alexsandro, Antônio, Arton, Bruno, Caio, Fernando, Gabriel, Ian e Joanderson cursam SO; [dados fictícios]
- Britney, Camila, Luana, Maiara, Mikaelle e Natália cursam TEES; [dados fictícios]
- Kelvyn, Lucas, Luiz, Marcelo, MateusViana, MateusSilva, Thiago, Victor e Wilton cursam TEBD; [dados fictícios]
- PP e SO são disciplinas obrigatórias;
- TEES e TEBD são disciplinas eletivas;
- Janderson leciona PP (Paradigmas de Programação);
- Janderson leciona SO (Sistemas Operacionais);
- Sabrina leciona TEES (Tópicos Especiais em Engenharia de Software);
- Vladimir leciona TEBD (Tópicos Especiais em Banco de Dados);
- Fulano é professor de Sicrano uma vez que Sicrano cursa uma disciplina que Fulano leciona.

Com base nisso:

i. Liste todas as disciplinas obrigatórias.
ii. Liste todos os alunos de Janderson.
iii. Liste todos os teus professores.
iv. Liste todos os professores de disciplinas eletivas.

Código:

```
aluno(alexsandro, pp).
aluno(antonio, pp).
aluno(arton, pp).
aluno(britney, pp).
aluno(bruno, pp).
aluno(caio, pp).
aluno(camila, pp).
aluno(fernando, pp).
aluno(gabriel, pp).
aluno(ian, pp).
aluno(joanderson, pp).
aluno(kelvyn, pp).
aluno(luana, pp).
aluno(lucas, pp).
aluno(luiz, pp).
aluno(maiara, pp).
aluno(marcelo, pp).
aluno(mateusViana, pp).
aluno(mateusSilva, pp).
aluno(mikaelle, pp).
```

```
aluno(natalia, pp).
aluno(thiago, pp).
aluno(victor, pp).
aluno(wilton, pp).

aluno(alexsandro, so).
aluno(antonio, so).
aluno(arton, so).
aluno(bruno, so).
aluno(caio, so).
aluno(fernando, so).
aluno(gabriel, so).
aluno(ian, so).
aluno(joanderson, so).

aluno(britney, tees).
aluno(camila, tees).
aluno(luana, tees).
aluno(maiara, tees).
aluno(mikaelle, tees).
aluno(natalia, tees).

aluno(kelvyn, tebd).
aluno(lucas, tebd).
aluno(luiz, tebd).
aluno(marcelo, tebd).
aluno(mateusViana, tebd).
aluno(mateusSilva, tebd).
aluno(thiago, tebd).
aluno(victor, tebd).
aluno(wilton, tebd).

disciplina(pp, obrigatoria).
disciplina(so, obrigatoria).
disciplina(tees, eletiva).
disciplina(tebd, eletiva).

leciona(janderson, pp).
leciona(janderson, so).
leciona(sabrina, tees).
leciona(vladimir, tebd).

professor(X, Y) :- aluno(Y, Z), leciona(X, Z).
```

Terminal:

i.

```
?- findall(X, disciplina(X, obrigatoria), Obrigatorias).
Obrigatorias = [pp, so].
```

ii.

```
?- findall(X, professor(janderson, X), Alunos_Janderson).
Alunos_Janderson = [alexsandro, antonio, arton, britney, bruno, caio, camila,
fernando, gabriel|...].
```

iii.

```
?- findall(X, professor(X, luiz), Professores_de_Luiz).
Professores_de_Luiz = [janderson, vladimir].
```

iv.

```
?- findall(X, (leciona(X, Z), disciplina(Z, eletiva)), Professores_Eletivas).
Professores_Eletivas = [sabrina, vladimir].
```

b) Baseando-se nos slides 89 a 97, utilize Prolog, via Java, considerando que o arquivo teste.pl deve consistir dos fatos e regras da ***letra a***. O programa em Java deve permitir ao usuário digitar o nome de um aluno, listando, por conseguinte, seus professores (***letra a - item iii***).
SUGESTÃO: Para pedir o nome, use `JOptionPane.showInputDialog("Nome?").`
OBS. 1: Trocar `ancestral(X, jose)` pelo que foi solicitado!
OBS. 2: Não esqueça de adicionar o jpl.jar ao Java Project.
OBS. 3: Em vez de `import jpl.*`, utilize: `import org.jpl7.*`.
OBS. 4: Em vez de `q1.query()`, utilize: `q1.hasSolution()`.
OBS. 5: Em vez de `Hashtable[]`, utilize: `Map<String, Term>[ ]`.
OBS. 6: Ao se deparar com esta possível exceção `java.lang.UnsatisfiedLinkError: no jpl in java.library.path`, consulte o link a seguir:
https://stackoverflow.com/questions/12283471/jpl-swi-prolog-configuration-failure
(Veja a resposta: "Try adding your path to java.library.path via Run > Run Configuration > [...]".)

Código:

```
package teste;

import java.util.Map;
import java.util.Scanner;

import org.jpl7.Atom;
import org.jpl7.Query;
import org.jpl7.Term;

public class Main{
	public static void main(String[] args){
		Query q1 = new Query("consult", new Term[] {new Atom("D:\\teste.pl")});
		System.out.println("consult " + (q1.hasSolution() ? "succeeded" : "failed"));
		Scanner sc = new Scanner(System.in);
		String nome = sc.nextLine();
		Query q2 = new Query(String.format("professor(X, %s)", nome));
		Map<String, Term>[] solution = q2.allSolutions();
		if (solution != null){
			for (int i = 0; i < solution.length; i++) {
				System.out.println("X = " + solution[i].get("X"));
			}
		}
	}
}
```

Terminal:

Exemplo 1

```
consult succeeded
luiz
X = janderson
X = vladimir
```

Exemplo 2

```
consult succeeded
camila
X = janderson
X = sabrina
```

2) Utilizando **Haskell**...

a) Baseando-se no slide 17, elabore um algoritmo em Haskell para indicar, a partir da frequência (porcentagem de faltas) e das notas das duas unidades de um determinado aluno, se este está 'Reprovado por falta', 'Reprovado por nota', 'Aprovado por média' ou 'Na final'.

Código:

```
main :: IO()

media nota1 nota2 = ((nota1 + nota2)/2)
faltas aulas freq = ((aulas - freq)/aulas * 100)

main = do
            putStrLn "Informe a nota da Unidade 1"
            nota1 <- readLn
            putStrLn "Informe a nota da Unidade 2"
            nota2 <- readLn
            putStrLn "Média"
            print (media nota1 nota2)

            putStrLn "Informe a quantidade total de aulas"
            aulas <- readLn
            putStrLn "Informe a frequencia"
            freq <- readLn
            putStrLn "Percentual de Faltas"
            print (faltas aulas freq)

            if ((faltas aulas freq) <= 25.0)
                then if ((media nota1 nota2) < 4.0)
                    then putStrLn "Reprovado por Nota"
                    else if ((media nota1 nota2) >= 7.0)
                        then putStrLn "Aprovado por Média"
                        else putStrLn "Está na Final"
                else putStrLn "Reprovado por Falta"
```

Terminal:

Exemplo 1

```
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking main ...
Informe a nota da Unidade 1
7.5
Informe a nota da Unidade 2
8
Média
7.75
Informe a quantidade total de aulas
20
Informe a frequencia
17
Percentual de Faltas
15.0
Aprovado por Média
```

Exemplo 2

```
Linking main ...
Informe a nota da Unidade 1
7.7
```

Informe a nota da Unidade 2
5.5
Média
6.6
Informe a quantidade total de aulas
20
Informe a frequencia
16
Percentual de Faltas
20.0
Está na Final

Exemplo 3

Linking main ...
Informe a nota da Unidade 1
7.7
Informe a nota da Unidade 2
6.6
Média
7.15
Informe a quantidade total de aulas
20
Informe a frequencia
8
Percentual de Faltas
60.0
Reprovado por Falta

Exemplo 4

Linking main ...
Informe a nota da Unidade 1
4.5
Informe a nota da Unidade 2
2.7
Média
3.6
Informe a quantidade total de aulas
20
Informe a frequencia
19
Percentual de Faltas
5.0
Reprovado por Nota

b) Procure como seria a implementação do algoritmo *Quicksort* em Haskell, entenda o código e modifique-o para ordenar uma lista, de maneira decrescente, considerando apenas os números pares. Teste-o "ordenando" a seguinte lista: [9,1,8,2,5,7,3,6,4]. Resultado esperado: [8,6,4,2]

Código:

```
main :: IO()
main = print (pares (quicksort [9,1,8,2,5,7,3,6,4]))

quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort list = quicksort right_sublist
                 ++ pivot_list
                 ++ quicksort left_sublist
        where
            pivot :: Int
            pivot = head list

            left_sublist :: [Int]
            left_sublist = filter (<pivot) list

            pivot_list :: [Int]
            pivot_list = filter (==pivot) list

            right_sublist :: [Int]
            right_sublist = filter (>pivot) list

pares [] = []
pares (x:xs)
        | x `mod` 2 /= 0 = pares xs
        | otherwise      = x:(pares xs)
```

Terminal:

```
Linking main ...
[8,6,4,2]
```

3) Considere Python uma linguagem de programação multiparadigma (incluindo características da Programação Funcional). Mostre duas formas de se implementar o algoritmo *Quicksort* em Python: de maneira imperativa (como tradicionalmente se aborda em uma disciplina de Estrutura de Dados) e de maneira funcional (similarmente ao que você procurou para a questão ***2b***, podendo ser codificado em uma linha!). Use a mesma sequência de elementos da questão ***2b*** para ilustrar a execução de ambas as formas. Comente sobre os códigos (em relação àqueles critérios comentados no início da disciplina: *readability* e *writability*).

a) Quicksort de maneira imperativa:

Código:

```
def quicksort(l):
    if l:
            left = [x for x in l if x < l[0]]
            right = [x for x in l if x > l[0]]
            if len(left) > 1:
                    left = quicksort(left)
            if len(right) > 1:
                    right = quicksort(right)
            return left + [l[0]] * l.count(l[0]) + right
    return []

A = [9, 1, 8, 0, 2, 5, 7, 10, 3, 6, 11, 4, 12]

print(A)
print(quicksort(A))
```

Terminal:

```
[9, 1, 8, 0, 2, 5, 7, 10, 3, 6, 11, 4, 12]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

b) <u>Quicksort de maneira funcional:</u>

<u>Código:</u>

```
q = lambda s:s if len(s)<2 else q([x for x in s[1:]if x<s[0]])+[s[0]]+q([x
for x in s[1:]if x >= s[0]])
A = [9, 1, 8, 0, 2, 5, 7, 10, 3, 6, 11, 4, 12]
print (A)
print (q(A))
```

<u>Terminal:</u>

```
[9, 1, 8, 0, 2, 5, 7, 10, 3, 6, 11, 4, 12]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Para ambos os códigos de ordenação usando Quicksort, temos o mesmo resultado e desempenho, entretanto, sob o ponto de vista de redigibilidade e legibilidade (writability and readability) o modo imperativo é melhor, visto que mesmo mais extensos, é mais fácil de observar seus parâmetros e variações, enquanto no modo funcional esta visualização não é intuitiva, e por muitas vezes, escapa aos olhos, levando a uma programação bem mais trabalhosa, embora mais concisa.

4) Comente brevemente sobre...

a) Programação Orientada a Aspectos (POA) e a linguagem AspectJ.

A programação orientada a aspectos é uma abordagem que permite a separação dos componentes funcionais de uma forma concisa, utilizando-se de mecanismos de abstração e de composição. O termo "aspecto" se refere aquilo que faz parte da aparência de algo.

O objetivo da programação orientada a aspectos é oferecer suporte para o desenvolvedor na tarefa de separar componentes entre si e os aspectos entre si, utilizando-se de mecanismos que permitam a abstração e composição destas.

Como esse paradigma geralmente envolve conceitos que independem da camada e não tem relação direta com os requisitos funcionais de um sistema, temos alguns benefícios como a reutilização, clareza e desacoplamento do código.

A programação orientada a aspectos também possibilita fazer a devida separação de responsabilidades, considerando funcionalidades que são essenciais para um grupo de objetos, mas não são de responsabilidade direta deles. São elementos básicos da Orientação a Aspectos:

- ***<u>Componente</u>**: unidade funcional expressa em uma linguagem de componentes; Propriedades de um sistema, no qual a implementação pode ser encapsulada de forma limpa em um procedimento generalizado.*
- ***<u>Aspecto (aspect)</u>**: unidade não funcional expressa em uma linguagem de aspectos; propriedades de um sistema, no qual a implementação não pode ser encapsulada em um procedimento generalizado.*
- ***<u>Crosscuting</u>**: entrelaçamento entre os interesses de domínio e os interesses ortogonais ou transversais como já falamos.*
- ***<u>Pontos de junção (join points)</u>**: são as partes do meu sistema que precisam de controle transacional.*
- ***<u>Conjuntos de junção (point cuts)</u>**: elementos da semântica da linguagem de componentes com os quais os programas de aspectos coordenam; elementos compostos por pontos de junção que tem por objetivo reunir informações a respeito do contexto dos pontos selecionados. Ou seja, point cuts são expressões utilizadas para identificar os join points.*
- ***<u>Advices</u>**: São os métodos que serão executados nos join points em um determinado momento a ser escolhido para execução.*
- ***<u>Regras de junção</u>**: código relativo aos requisitos ortogonais que deve ser executado nos pontos de junção.*
- ***<u>Processo de combinação (weaving)</u>**: composição entre os componentes e os aspectos.*

- ***Combinador de aspectos (aspect weaver)****: processador de linguagem especial que oferece suporte à composição entre os componentes e os aspectos.*

A orientação a aspectos não é um substituto a orientação a objetos e sim um complemento para diminuição do acoplamento dos módulos possibilitando mais facilidade de manutenção, maior reaproveitamento do código e maior organização do sistema. Tanto que a primeira implementação da Programação Orienta a Aspectos, o AspectJ para a linguagem Java, surgiu na década de 80.

AspectJ é uma extensão orientada a aspectos, de propósito geral, da linguagem Java. A principal construção em AspectJ é o aspecto, o qual define uma função específica que pode afetar várias partes de um sistema, como, por exemplo, distribuição. Um aspecto, como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, através da definição de aspectos especializados

b) Programação Concorrente (baseando-se no capítulo 17 do livro do Tucker (2010), e/ou em outra referência relevante).

A programação concorrente pode ocorrer em muitos níveis de linguagem – desde o nível mais baixo de lógica digital até o nível de aplicação. A concorrência também ocorre nos quatro paradigmas de programação – imperativa, orientada a objetos, funcional e lógica.

A introdução da concorrência em um programa pode poupar uma quantidade enorme de recursos de computação, tanto em espaço quanto em velocidade. Sem dúvida, muitas aplicações importantes da computação não seriam concebíveis se a concorrência não fosse um elemento central de seus projetos. No entanto, a programação concorrente em qualquer nível ou domínio de aplicação traz complexidades especiais e fundamentais.

O termo concorrência representa um programa em que dois ou mais contextos de execução podem estar ativos simultaneamente. Programas com um único contexto de execução são chamados de programas singlethreaded. Um programa com múltiplos contextos de execução é chamado de multi-threaded. Em um programa multi-threaded, parte do estado do programa é compartilhado entre as threads, enquanto parte do estado (incluindo o controle de fl uxo) é único para cada thread.

Tanto em Java quanto em Ada, uma thread está associada a um método separado (função), em vez de a uma única instrução. No entanto, para iniciar uma thread são necessárias ações adicionais, além de chamar uma função. Primeiro, quem chama uma nova thread não espera que ela se complete antes de continuar; ele passa à execução das instruções que seguem a chamada da thread. Segundo, quando a thread termina, o controle não retorna para quem a chamou.

Programas concorrentes requerem comunicação ou interação interthreads. A comunicação ocorre pelas seguintes razões:

- *Uma thread, às vezes, requer acesso exclusivo a um recurso compartilhado, como uma fila de impressão, por exemplo, uma janela de um terminal ou um registro em um arquivo de dados.*
- *Uma thread, às vezes, precisa trocar dados com outra thread.*

Em ambos os casos, as duas threads em comunicação devem sincronizar sua execução para evitar conflito ao adquirir recursos ou para fazer contato ao trocar dados. Uma thread pode se comunicar com outras threads por meio de:

- *Variáveis compartilhadas: esse é o mecanismo primário usado em Java, e também pode ser usado em Ada.*
- *Passagem de mensagens: esse é o mecanismo primário usado em Ada.3*
- *Parâmetros: são usados em Ada em conjunto com a passagem de mensagens.*

Threads normalmente cooperam umas com as outras para resolver um problema. No entanto, é altamente desejável manter a comunicação entre threads em um nível mínimo; isso torna o código mais fácil de entender e deixa cada thread rodar em sua própria velocidade sem ser retardada por protocolos complexos de comunicação.

c) Teoria de domínios (relacionado ao conteúdo sobre Semântica Denotacional).

Na teoria das linguagens de programação, a semântica é o campo que se preocupa com o estudo matemático rigoroso do significado das linguagens. Ele faz isso avaliando o significado de entradas sintaticamente válidas definidas por uma linguagem de programação específica, mostrando a computação envolvida. Nesse caso, em que a avaliação seria de entradas sintaticamente inválidas, o resultado seria o não cálculo.

A semântica descreve os processos que um computador segue ao executar um programa naquele idioma específico. Isso pode ser mostrado descrevendo a relação entre a entrada e saída de um programa, ou uma explicação de como o programa será executado em uma determinada plataforma, criando assim um modelo de computação. O campo da semântica formal abrange os seguintes pontos:

- *A definição de modelos semânticos*
- *As relações entre os diferentes modelos semânticos*
- *As relações entre as diferentes abordagens do significado*
- *A relação entre a computação e as estruturas matemáticas subjacentes de campos como lógica, teoria dos conjuntos, teoria do modelo, teoria das categorias, etc.*

Ele tem ligações estreitas com outras áreas da ciência da computação, como design de linguagem de programação, teoria dos tipos, compiladores e interpretadores, verificação de programas e verificação de modelos. Existem muitas abordagens para a semântica formal; estes pertencem a três classes principais: Semântica Denotacional, Operacional e Axiomática.

Semântica denotacional, é aquela que em que cada frase da língua é interpretada como uma denotação, ou seja, um significado conceitual que pode ser pensado abstratamente. Essas denotações são frequentemente objetos matemáticos que habitam um espaço matemático, mas não é obrigatório que o sejam. Como uma necessidade prática, as denotações são descritas usando alguma forma de notação matemática, que por sua vez pode ser formalizada como uma metalinguagem denotacional. Por exemplo, a semântica denotacional de linguagens funcionais frequentemente traduz a linguagem em teoria de domínio. As descrições semânticas denotacionais também podem servir como traduções composicionais de uma linguagem de programação para a metalinguagem denotacional e usadas como base para projetar compiladores.