Prova Final de Linguagens de Programação
- DCC024B -
Ciência da Computação

Nome: _____
"Eu dou minha palavra de honra que não trapacearei neste exame."

Número de matrícula: _____

As regras do jogo:

- A prova é sem consulta.
- Quando terminar, não entregue nada além do caderno de provas para o instrutor.
- Quando escrever código, a sintaxe correta é importante.
- Cada estudante tem direito a fazer uma pergunta ao instrutor durante a prova. Traga o caderno de provas quando vier à mesa do instrutor.
- A prova termina uma hora e quarenta minutos após seu início.

Alguns conselhos:

- Escreva sempre algo nas questões, a fim de ganhar algum crédito parcial.
- Se não entender a questão, e já tiver gasto sua pergunta, escreva a sua interpretação da questão junto à resposta.
- Serão avaliadas as seis melhores respostas. Então sinta-se livre para abandonar alguma questão devido ao tempo.
- A prova não é difícil, ela é divertida, então aproveite!

Tabela 1: Pontos acumulados

Questão 1	Questão 2	Questão 3	Questão 4	Questão 5	Questão 6	Questão 7

1. Uma linguagem é dita reflexiva quando programas escritos nesta linguagem podem obter ou modificar informações a respeito de si mesmos.

   (a) (5 pontos) A reflexividade *passiva* ocorre quando a linguagem de programação permite que o programa descubra informações a respeito de si mesmo. Escreva um programa, em pseudo-código, que ilustre este conceito. Este programa pode ser escrito em um paradigma imperativo, lógico ou funcional. Explique qual o papel da reflexividade em seu exemplo, e informe o nome de uma linguagem de programação onde seja possível escrever um programa com capacidade similar.

   (b) (5 pontos) A reflexividade é dita *ativa* quando o programa pode modificar-se. Novamente, escreva um programa, em pseudo-código, que ilustre tal conceito. Explique como, neste caso, o programa se modifica, e informe o nome de uma linguagem de programação que permita a criação de um programa similar.

2. Esta questão refere-se à passagem de parâmetros por nome. Este método causa a avaliação do parâmetro real sempre, e somente se, ele for utilizado no corpo da função chamada.

   (a) (5 pontos) Para evitar a captura de variáveis, um fenômeno que ocorre na passagem de parâmetros por expansão de macros, a avaliação do parâmetro formal, na passagem por nomes, utiliza o contexto do chamador. Escreva um programa, em pseudo-código, que retornaria resultados diferentes, dependendo do método de chamada ser expansão de macros ou passagem por nome.

   (b) (5 pontos) A passagem por nomes difere da avaliação preguiçosa por que esta última utiliza um cache para evitar a múltipla avaliação de parâmetros. Escreva um programa, em pseudo-código, que mostre de forma irrefutável a diferença entre estes dois mecanismos de passagem de parâmetros.

3. A questão que se segue utiliza à interface e às duas classes abaixo, que implementam uma lista encadeada. Um nodo pode conter um dado, e um próximo nodo, ou ele pode ser um nodo nulo, que indica o final da lista. Nodos são criados assim: `Node<Integer> n1 = new ConsNode<Integer>(0, new NilNode<Integer>());`. Em cada um dos items, você deverá implementar um método tanto para a classe `ConsNode` quanto para a classe `NilNode`.

```
interface Node<E> {
  boolean contains(E e);
  void replace(E oldE, E newE);
  Node<E> append(Node<E> n);
}
```

```
class ConsNode<E> implements Node<E> {
  ConsNode(E elem, Node<E> n) { ... }
  public boolean contains(E e) { ... }
  public void replace(E oldE, E newE) { ... }
  public Node<E> append(Node<E> n) { ... }
}
```

```
class NilNode<E> implements Node<E> {
  public boolean contains(E e) { ... }
  public void replace(E oldE, E newE) { ... }
  public Node<E> append(Node<E> n) { ... }
}
```

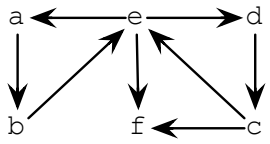
(a) (3 pontos) Implemente o método `contains` em ambas as classes. Este método retorna verdadeiro se o nodo contém um dado elemento, ou um de seus sucessores contêm tal elemento.

(b) (3 pontos) Implemente o método `replace`, que recebe dois elementos, um antigo e um novo, e substitui todas as ocorrências do elemento antigo pelo elemento novo.

(c) (4 pontos) Implemente o método `append`, tal que `n.append(l)` vai inserir `l` como o último elemento na cadeia de nodos que começa com `n`. Qual o comportamento de `n.append(n)` em sua implementação?

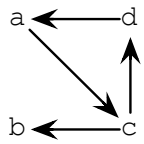
4. (10 pontos) Em Prolog podemos representar um digrafo via uma lista de vértices, e um conjunto de predicados `edge(V1, V2)`, que é verdade se o grafo possuir uma aresta do nodo `V1` para o nodo `V2`. Abaixo temos dois exemplos de grafos:

```
vertices([a, b, c, d, e, f])

edge(a, b).
edge(b, e).
edge(e, d).
edge(d, c).
edge(c, e).
edge(e, f).
edge(e, a).
edge(c, f).
```



(Grafo G1)

```
vertices(a, b, c, d])

edge(c, b).
edge(a, c).
edge(c, d).
edge(d, a).
```



(Grafo G2)

Dado um programa que represente um grafo, implemente um predicado `hamilPath(L)`, que é verdade se a lista `L` for uma permutação da lista de vértices que designe um *caminho hamiltoniano* neste grafo. Um caminho hamiltoniano é uma sequência de vértices adjacentes que contenha todos os vértices do dígrafo, mas que contenha cada vértice somente uma vez. Em nosso exemplo, teríamos:

```
?- hamilPath(G1)
G1 = [a, b, c, d, e, f];
...

?- hamilPath(G2)
false.
```

Assuma que seu programa contenha um predicado `vertices[L]`, onde `L` é uma lista de vértices, e uma série de predicados `edge(u, v)`, onde tanto `u` quanto `v` sejam átomos presentes em `L`. **Oh, e esqueça a eficiência: este problema é NP-completo**.

5. Linguagens ditas *Orientadas por Objetos* são conhecidas por utilizarem a memória *heap* pesadamente. O *heap* é muito mais difícil de gerir que a alocação estática, e mesmo a alocação de pilha. Entretanto, há situações em que dados do *heap* podem ser movidos para as outras áreas de memória, mais "manuseáveis".

   (a) (3 pontos) Em Java, objetos normalmente são armazenados no *heap*. Escreva um programa em Java que não funcionaria de acordo com a semântica padrão da linguagem caso objetos fossem alocados na pilha, e não no *heap*. Por que o programa não funcionaria corretamente?

   (b) (4 pontos) Muitos compiladores implementam uma análise de código chamada *Escape Analysis*. Esta análise prova que um objeto, criado no corpo de um método, pode ser alocado na pilha, sem que isto mude a semântica do programa. Descreva, de forma objetiva, os critérios que o compilador deveria utilizar para provar que um objeto pode ser alocado na pilha.

   (c) (3 pontos) Qual a vantagem de se alocar um objeto na pilha, e não no *heap*, como normalmente é feito?

6. Na linguagem Fortran, arranjos são normalmente alocados em memória segundo o modelo *column-major*. Isto quer dizer que matrizes são alocadas por coluna, conforme mostrado na figura abaixo:

Uma matriz como esta: ... ... será alocada em memória assim:

```
00   01   02        ... 00 01 02 10 11 12 20 21 22 ...
10   11   12
20   21   22
```

(a) (3 pontos) Qual é a fórmula para calcular o endereço, em bytes, de um arranjo bidimensional `A[i, j]` contendo `M` linhas e `N` colunas? Ao escrever a fórmula, assuma que o endereço da primeira posição do arranjo é dada por `base`, e que cada palavra da arquitetura alvo contém `word` bytes.

(b) (7 pontos) Existe alguma provável diferença de eficiência entre os dois programas abaixo, escritos em Fortran 90? Justifique a sua resposta.

(i) Fixa-se a linha, varia-se a coluna:

```
real A(3000,3000)
integer i,j
j = 1
i = 1
while (i < 3000) do
  while (j < 3000) do
    a(i,j) = real(i)/real(j)
  enddo
enddo
```

(ii) Fixa-se a coluna, varia-se a linha:

```
real A(3000,3000)
integer i,j
j = 1
i = 1
while (j < 3000) do
  while (i < 3000) do
    a(i,j) = real(i)/real(j)
  enddo
enddo
```

7. (10 pontos) Considere o programa Java abaixo, e diga o que será impresso pelo método `main`.

```
public class Avatar {
  public void buy(Knife k) {
    System.out.println("Avatar bought a knife");
  }
  public void buy(Sword s) {
    System.out.println("Avatar bought a sword");
  }
}

public class Knife {
  public void isBoughtBy(Avatar a) {
    a.buy(this);
  }
}

public class Sword extends Knife {
  public void isBoughtBy(Avatar a) {
    a.buy(this);
  }
}

public class SpiderAv extends Avatar {
  public void buy(Knife k) {
    System.out.println("Spider bought a knife");
  }
  public void buy(Sword s) {
    System.out.println("Spider bought a sword");
  }
  public static void main(String args[]) {
    Avatar a1 = new Avatar();
    Avatar a2 = new SpiderAv();
    SpiderAv sa = new SpiderAv();
    Knife ks = new Sword();
    a1.buy(ks);                          // 1 Ponto
    a2.buy(ks);                          // 1 Ponto
    sa.buy(ks);                          // 2 Pontos
    ks.isBoughtBy(a1);                   // 2 Pontos
    ks.isBoughtBy(a2);                   // 2 Pontos
    ks.isBoughtBy(sa);                   // 2 Pontos
  }
}
```