Prova Final de Linguagens de Programação
- DCC024 -
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.
- Seja honesto e lembre-se: **você deu sua palavra de honra**.

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.
- Lembre-se do ganhador do Oscar de melhor filme em 2004. E lembre-se também: perguntando qual foi o filme ganhador você perde a sua pergunta.
- A prova não é difícil, ela é divertida, então aproveite!

Tabela 1: Pontos acumulados (para uso do instrutor)

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

1. Existem linguagens em que a indentação é parte da sintaxe. Python é uma destas linguagens. Uma outra linguagem assim foi a primeira versão de Fortran.

   (a) (2 pontos) Considere, por exemplo, os dois programas abaixo, ambos escritos em Python:

```
def fact1(n):
  ans = 1
  while n > 1:
    ans *= n
    n -= 1
  return ans
```

```
def fact2(n):
  ans = 1
  while n > 1:
    ans *= n
  n -= 1
  return ans
```

   O programa da esquerda calcula o fatorial de um inteiro `n`. O que faz o programa da direita?

   (b) (4 pontos) Por que *Guido van Rossum*, o "pai"de Python, optou por tornar a indentação importante para definir a semântica dos programas?

   (c) (4 pontos) A indentação também era parte da sintaxe das primeiras versões de Fortran, implementadas no final da década de 50. Porém, naquele caso, a razão desta importância da indentação era bem diferente daquela observada em Python. Por que a indentação era parte da sintaxe das primeiras versões de Fortran?

2. Muitas linguagens orientadas à objetos possuem o conceito de *classes*. Classes permitem que desenvolvedores implementem tipos abstratos de dados. Java e Python são duas linguagens que possuem o conceito de classes. Por exemplo, abaixo temos uma classe `Animal` implementada em ambas as linguagens:

Java	Python

`class Animal {`
`  public void eat() {`
`  System.out.println("eating");`
`  }`
`  public String toString () {`
`  return "Animal";`
`  }`
`}`
	`class Animal:`
	`  def __init__(self, name):`
	`  self.name = name`
	`  def __str__(self):`
	`  return "Animal"`
	`  def eat(self):`
	`  print "eating."`

(a) (6 pontos) Embora ambas as linguagens possuam o conceito de classe, elas são implementadas de forma muito diferente nestas linguagens. Preencha a tabela abaixo com um dos seguintes símbolos $\emptyset$ (nenhuma linguagem), J (Java), P (Python) ou JP (Java e Python), dependendo de cada uma destas linguagens disponibilizar, em sua implementação de classes, a funcionalidade em questão ou não.

Funcionalidade	Linguagens que a apresenta
Herança múltipla	
Sobrecarga de métodos nas classes herdeiras	
Classes como valores de "primeira classe"	
Controle de visibilidade	
Chamada de métodos em tempo $O(1)$	
Inserção de novos métodos durante a execução	

(b) (4 pontos) Em geral, tanto Java quanto Python vão usar pesadamente a área de alocação de dados conhecida como *heap*. Pelo menos estas linguagens tendem a usar o heap mais que C ou Pascal, por exemplo. Por que estas linguagens tendem a usar tanto o heap?

3. O programa abaixo, implementado em C, contém duas funções, `swap_ii` e `swap_dd`, que trocam o conteúdo de variáveis inteiras e de ponto flutuante, respectivamente.

```
void swap_ii(int* a, int* b) {
  int __tmp = *a;
  *a = *b;
  *b = __tmp;
}
void swap_dd(double* a, double* b) {
  int __tmp = *a;
  *a = *b;
  *b = __tmp;
}
```

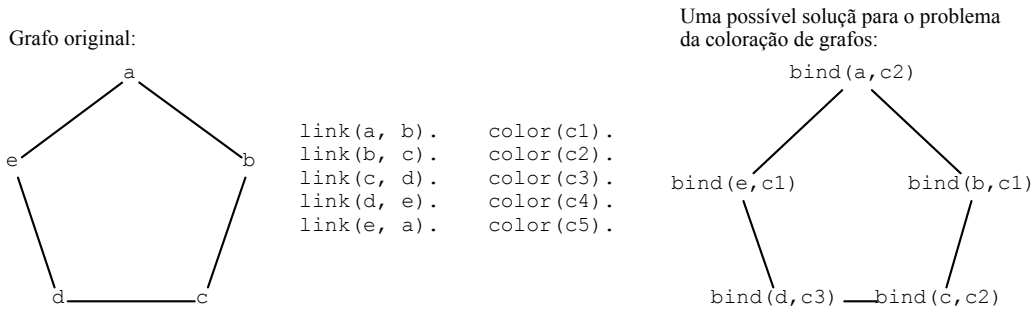
```
int main() {
  int i1 = 2, i2 = 3;
  double d1 = 2.71, d2 = 3.14;
  swap_ii(&i1, &i2);
  swap_dd(&d1, &d2);
  printf("i1=%d, i2=%d\n", i1, i2);
  printf("d1=%lf, d2=%lf\n", d1, d2);
}
```

O código deste programa é um tanto quanto redundante: as funções `swap_ii` e `swap_dd` possuem quase o mesmo código, exceto pelo tipo dos parâmetros. Se a linguagem C possuísse alguma forma de polimorfismo paramétrico, então talvez pudéssemos mesclar ambas as funções em uma implementação comum. C não provê este tipo de polimorfismo, porém esta linguagem, que não nos cessa de encantar, possui *macros* – um recurso com o qual podemos aumentar a sua sintaxe.

(a) (5 pontos) Implemente uma macro `SWAP` que receba três coisas: os dois parâmetros que precisam ser trocados, mais o tipo destes parâmetros. `SWAP` deve ser capaz de implementar tanto a funcionalidade de `swap_ii` quanto `swap_dd`, ou qualquer outra função de troca de variáveis que se pareça com as duas acima.

(b) (5 pontos) Re-implemente a função `main` acima, para que ela utilize duas chamadas à sua macro `SWAP` em vez das funções `swap_ii` e `swap_dd`.

4. (10 pontos) O problema da coloração de grafos é um clássico problema NP-completo: temos um conjunto de cores C, mais um grafo $G = (V, E)$, formado por um conjunto de vértices V e um conjunto de arestas E. O problema pede por uma associação entre elementos de C e vértices em V tal que dados dois vertices $v_1 \in V$ e $v_2 \in V$, se a aresta $v_1v_2 \in E$, então a cor associada a v_1 é diferente da cor associada à v_2. Em Prolog podemos representar este problema como mostrado na figura abaixo:




```
sol([bind(a, C1), bind(b, C2), bind(c, C3), bind(d, C4), bind(e, C5)]) :-
  color(C1), color(C2), color(C3), color(C4), color(C5),
  \+ invalid([bind(a, C1), bind(b, C2), bind(c, C3), bind(d, C4), bind(e, C5)]).
```

O predicado `link(X, Y)` é usado para representar as arestas do grafo. Ele é verdade se o vértice `X` for conectado ao vértice `Y`. Já o predicado `color(c)` é verdade se o átomo `c` for um elemento do conjunto de cores C. O predicado `sol(L)` é verdade se a lista `L` for uma solução para o problema da coloração de grafos. Cada elemento desta lista é um átomo composto `bind(v, c)`, que denota o fato de a cor `c` estar associada ao vértice `v`. Nesta questão você deve implementar o predicado `invalid(L)`, que é verdade se a lista `L` **não** for uma solução válida para o problema da coloração de grafos. Uma solução é invalida se existe uma cor `c` associada vértices `v1` e `v2` adjacentes no grafo. Em Prologuês, a solução é invalida se `L` contém dois elementos `bind(v1, c)` e `bind(v2, c)`, tal que o predicado `link(v1, v2)` seja verdadeiro. Não se preocupe com a eficiência de sua implementação. Caso você se lembre dos predicados `member` e `select`, então você consegue implementar o predicado `invalid` com somente duas linhas! Você não precisa implementar `member` ou `select`, pois estes predicados fazem parte da biblioteca padrão de Prolog, mas você deve saber a sintaxe correta para usá-los. Se preferir, sinta-se livre para implementar `invalid` sem usar estes predicados.

5. Considere os dois laços abaixo, retirados de um programa escrito em C:

(a)
```
for (int k = 0; k < Width * Width; k += Width) {
  Out[tid] += In[k] / (a - b) - In[k] / (c - d);
}
```

(b)
```
for (int k = 0; k < Width * Width; k += Width) {
  Out[tid] += In[k] / (a - b);
  Out[tid] -= In[k] / (c - d);
}
```

A tabela abaixo mostra o código *assembly* que `gcc` produz para ambos os programas.

Primeiro laço (a)	Segundo laço (b)

`$Lt_0_3074:`
`ld.global.f32 %f7, [%r12+0];`
`mov.f32 %f8, %f6;`
`sub.f32 %f9, %f8, %f5;`
`div.full.f32 %f10, %f7, %f9;`
`sub.f32 %f11, %f4, %f3;`
`div.full.f32 %f12, %f7, %f11;`
`sub.f32 %f13, %f10, %f12;`
`add.f32 %f1, %f1, %f13;`
`st.global.f32 [%r7+0], %f1;`
`setp.lt.u32 %p3, %r12, %r13;`
`@%p3 bra $Lt_0_3074;`
	`$Lt_0_3074:`
	`mov.f32 %f7, %f6;`
	`ld.global.f32 %f8, [%r12+0];`
	`sub.f32 %f9, %f7, %f5;`
	`div.full.f32 %f10, %f8, %f9;`
	`add.f32 %f11, %f1, %f10;`
	`st.global.f32 [%r7+0], %f11;`
	`ld.global.f32 %f12, [%r12+0];`
	`sub.f32 %f13, %f4, %f3;`
	`div.full.f32 %f14, %f12, %f13;`
	`sub.f32 %f1, %f11, %f14;`
	`st.global.f32 [%r7+0], %f1;`
	`setp.lt.u32 %p3, %r12, %r13;`
	`@%p3 bra $Lt_0_3074;`

Podemos ver que, quando compilando o laço (a), `gcc -O3` é capaz de usar somente uma instrução de leitura de memória para obter o valor de `In[k]`, ainda que este valor seja lido duas vezes no interior do laço. Esta instrução é `ld.global.f32 %f7, [%r12+0]`. Neste caso, `gcc -O3` lê o valor armazenado na posição `[%r12+0]`, deposita este valor no registrador `%f7`, e a partir daí passa a usar somente este registrador, cujo acesso é muito mais rápido que a memória.

Infelizmente `gcc -O3` não é capaz de otimizar o laço (b). Isto é, mesmo neste nível de otimização, `gcc` produz duas instruções de carregamento (`ld.global.f32`) para ler o valor de `In[k]`. Por que `gcc -O3` não pode remover a segunda leitura da memória no caso do laço (b), mantendo o valor lido da primeira vez em um registrador, conforme feito no primeiro laço?

6. Uma função recursiva é dita de *cauda rasa* quando a última coisa que ela faz é chamar-se recursivamente. Re-implemente cada um dos predicados abaixo, para que eles sejam predicados de cauda rasa:

   (a) (4 Pontos) O predicado que soma os elementos de uma lista:

```
sum([], 0).
sum([H|T], X) :- sum(T, XAux), X is XAux + H.
```

   (b) (4 Pontos) O predicado que inverte os elementos de uma lista:

```
myappend([], L, L).
myappend([H|T], L, [H|LAux]) :- myappend(T, L, LAux).

myreverse([], []).
myreverse([H|T], R) :- myreverse(T, RT), myappend(RT, [H], R).
```

   (c) (2 Pontos) Afinal de contas, qual a vantagem de implementarmos funções de cauda rasa?