Prova Final de Linguagens de Programação
- DCC024W -
Sistemas de Informação

Nome: _____

"Eu dou minha palavra de honra que não trapaceei, estou trapaceando, ou 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. O instrutor avisará quando faltarem somente 15 minutos para o final do exame.
- 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 da cor dos domingos. E lembre-se também: perguntando qual é a cor dos domingos 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. Essa questão refere-se ao programa abaixo, escrito na linguagem de programação Python:

```
def foo(l, e):
  glVar = 0
  for x in l:
    if x == e:
      break
  else:
    glVar = 1
  print glVar

foo([2, 3, 5, 7], 3)
foo([2, 3, 5, 7], 4)
```

   (a) (1 ponto) o que será impresso pelo programa acima?

   (b) (9 pontos) Escreva uma função `foo`, na linguagem C, que seja equivalente ao programa Python visto acima. A sua função deverá ler um arranjo `l`, o seu tamanho `N` e um inteiro `e`. Use a guia logo abaixo:

```
#include <stdio.h>

void foo(int* l, int N, int e) {
  int glVar = 0;
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
?
  printf("%d\n", glVar);
}

void main() {
  int a1[] = {2, 3, 5, 7};
  foo(a1, 4, 3);
  foo(a1, 4, 4);
}
```

2. Programadores cometem vários tipos de erros enquanto desenvolvem *software*. Alguns desses erros são descobertos em tempo de compilação. Outros são descobertos em tempo de execução. Finalmente, alguns erros nunca são descobertos, e o programa, tendo executado, pode ficar em um estado indefinido.

   (a) (2 Pontos) Escreva um programa, em C, que seja **sintaticamente** correto, mas que não seria compilado devido a algum erro de natureza estática.

   (b) (4 pontos) Escreva um programa em alguma linguagem de sua escolha, que é corretamente compilado, mas que gera um erro em tempo de execução.

   (c) (4 pontos) Escreva um programa em alguma linguagem de sua escolha, que é corretamente compilado e executa até terminar, mas que termina em um estado indefinido devido a um erro em tempo de execução. Note que esse tipo de comportamento somente é possível em linguagens **fracamente tipadas**.

3. Algumas linguagens de programação provêem o programador com uma facilidade chamada *reflexão estática*. Linguagens que possuem esse tipo de reflexão permitem que o tipo dos dados seja conhecido em tempo de execução. Python é uma dessas linguagens, conforme mostra o programa abaixo:

```
>>> class A:
...   def __init__(self, e):
...     self.e = e
...
>>> a = A(42)
>>> isinstance(a, A)
True
>>> isinstance(a, list)
False
```

   (a) (2 Pontos) A linguagem C apresenta ou não esse tipo de reflexão?

   (b) (8 Pontos) Se a sua resposta para a questão anterior houver sido afirmativa, complete o trecho de código abaixo, mostrando como imprimir o tipo do registro `struct bar`. Caso a sua resposta seja negativa, explique porque não é possível saber o tipo de `struct bar` em tempo de execução.

```
struct bar {
  int e;
};

int main() {
  struct bar a;
  a.e = 42;
?
?
?
?
?
?
?
?
?
  return 0;
}
```

4. Nessa questão você irá escrever, na linguagem Python, a função *rev*, que inverte uma lista, de duas formas diferentes.

   (a) (5 Pontos) Escreva uma função `imperative_rev(l)`, que receba uma lista `l` e a inverta *in-place*, isto é, sem a necessidade de replicar a lista. A sua função deve ter complexidade linear em tempo, e deve criar uma quantidade constante de dados auxiliares, caso esses sejam necessários. Um exemplo de execução é dado logo abaixo:

```
>>> def imperative_rev(l):
... # essa funcao precisa
... # ser implementada
...
>>> l = [2,3,5,7]
>>> imperative_rev(l)
>>> l
[7, 5, 3, 2]
```

   (b) (5 Pontos) Escreva uma função `functional_rev(l, acc)` que produza uma lista l' que é o inverso da lista de entrada `l`. A sua função não pode alterar a lista de entrada. Além disso, a sua função deve ter uma implementação de *cauda rasa*. Em outras palavras, ela precisa ser implementada de forma tal que a última ação que ela tome seja chamar-se recursivamente. Um exemplo de execução é dado logo abaixo:

```
>>> def functional_rev(l):
... # essa funcao precisa
... # ser implementada
...
>>> l = [2,3,5,7]
>>> l = functional_rev(l)
>>> l
[7, 5, 3, 2]
```

5. (Um ponto por cada item correto) A linguagem Algol foi uma das linguagens de programação que mais influenciou o projeto de outras linguagens. Assim, várias das características de Algol são hoje encontradas em linguagens modernas. Para cada uma das características de Algol listadas abaixo, escreva ao seu lado uma linguagem de programação que apresente a mesma característica. **Você pode reutilizar a mesma linguagem no máximo três vezes**.

   (a) Blocos delimitadores:

   (b) Estrutura léxica de formato livre:

   (c) Escopo estático:

   (d) Tipagem estática com anotações de tipo:

   (e) if-then-else's aninhados:

   (f) Passagem de parâmetros por valor:

   (g) Expressões condicionais:

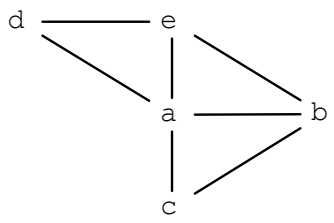
   (h) Procedimentos de primeira classe:

   (i) Operadores definidos pelo usuário:

   (j) Passagem de parâmetros por nome:

6. Uma *clique* é um grafo completo. O problema de decidir se um grafo G possui uma clique de tamanho N é um problema NP-completo bem conhecido. Esse problema, inclusive, é parte da lista de 21 problemas proposta por Richard Karp em 1972. Embora não conheçamos qualquer algoritmo eficiente para encontrar cliques em grafos, é muito fácil resolver esse problema por força bruta em Prolog. Nesse caso, podemos representar um grafo como um conjunto de arestas, conforme feito na figura abaixo:

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



Cliques:

```
a, b, c
a, b, e
a, e, d
```

O predicado `cliqueN`, definido abaixo, é verdade quando `G` é uma lista de vértices de um grafo, `N` é um número inteiro, e `L` é uma sublista de `G` que forma uma clique:

```
cliqueN(N, G, L) :- sublist(G, L), length(L, N), clique(L).
```

Por exemplo:

```
?- cliqueN(3, [a, b, c, d, e], L).
L = [a, b, c] ;
L = [a, b, e] ;
L = [a, d, e] ;
false.
```

(a) (4 Pontos) Defina o predicado `sublist(G, L)`, que seja verdade quando `L` for uma sublista de `G`.

(b) (6 Pontos) Defina o predicado `clique(L)`, que seja verdade quando `L` for uma lista de vértices que forme um grafo completo.