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

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**.

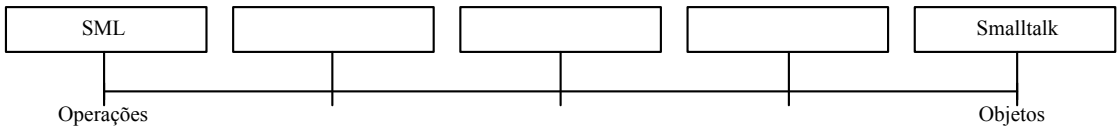
Alguns conselhos:

- 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. Algumas linguagens de programação favorecem um estilo de programação mais voltado às *operações* algorítmicas que compõem o problema a ser resolvido. SML é um exemplo de linguagem assim. Em SML é mais natural programarmos de forma *top-down*: um problema é resolvido por uma função principal, a qual deve ser expandida em várias sub-rotinas. Esse processo continua até que cada uma das sub-rotinas tenha sido implementada. Outras linguagens favorecem um estilo de programação mais voltado aos *tipos de dados*. Nessas linguagens as operações são parte dos tipos. Dados, nesse caso, "sabem"executar as ações. O desenvolvimento tende, então, a ser visto de forma *bottom-up*: o programa consiste em um conjunto de módulos. Cada módulo define um tipo, e cada tipo define as operações que ele sabe realizar. É claro que essa dimensão de classificação de linguagens: orientada a operações ou orientada a objetos não é rígida. O programador pode desenvolver programas seguindo uma filosofia, ainda que essa abordagem não seja a mais natural na linguagem que ele está usando. Além disso, algumas linguagens permitem tanto uma forma de desenvolvimento quanto a outra. Temos assim o espectro abaixo:



De um lado temos SML, uma linguagem muito voltada às operações, e do outro temos *Smalltalk*, uma linguagem muito voltada aos objetos.

(a) (1 Pontos) Preencha as caixas acima com as seguintes linguagens: Python, Java e C. A linguagem mais orientada às operações deve estar mais próxima de SML, e a linguagem mais orientada aos objetos deve estar mais próxima de Smalltalk.

(b) (3 Pontos) Justifique a sua escolha para Python.

(c) (3 Pontos) Justifique a sua escolha para Java.

(d) (3 Pontos) Justifique a sua escolha para C.

2. A linguagem *assembly* logo abaixo é *Turing* Completa:

(Variaveis)	::=	$\{v_1, v_2, \ldots\}$
– (Atribuição)	|	$\texttt{mov}(v_1, v_2)$
– (Adição)	|	$\texttt{add}(v_1, v_2, v_3)$
– (Armazenamento em memória)	|	$\texttt{stm}(v_0, v_1)$
– (Carregamento da memória)	|	$\texttt{ldm}(v_1, v_0)$
– (Desvio se zero)	|	$\texttt{bzr}(v, l)$
– (Desvio incondicional)	|	$\texttt{jmp}(l)$

Um programa nessa linguagem pode ser descrito por uma tupla de três elementos: $(P, \texttt{pc}, \Sigma)$. P é um arranjo contendo instruções assembly. O inteiro `pc` é um contador usado para indexar instruções em P. Finalmente, Σ é uma função que associa nomes de variáveis a valores inteiros. Algumas instruções alteram o valor de `pc`. Outras alteram Σ. Abaixo vemos a semântica natural de três dessas instruções: `add`, `stm` e `bzr`. A notação $f[a \mapsto b]$ denota $\lambda x.\mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f(x)$.

$$[\textsc{AddSem}]\quad \frac{P[\texttt{pc}] = \texttt{add}(v_1, v_2, v_3) \qquad \Sigma[v_2] = n_2 \qquad \Sigma[v_3] = n_3 \qquad \Sigma' = \Sigma[v_1 \mapsto (n_2 + n_3)] \qquad \langle P, \texttt{pc} + 1, \Sigma' \rangle \to \Sigma''}{\langle P, \texttt{pc}, \Sigma \rangle \to \Sigma''}$$

$$[\textsc{StmSem}]\quad \frac{P[\texttt{pc}] = \texttt{stmem}(v_0, v_1) \qquad \Sigma[v_0] = x \qquad \Sigma[v_1] = n \qquad \Sigma' = \Sigma[x \mapsto n] \qquad \langle P, \texttt{pc} + 1, \Sigma' \rangle \to \Sigma''}{\langle P, \texttt{pc}, \Sigma \rangle \to \Sigma''}$$

$$[\textsc{BzrSem}]\quad \frac{P[\texttt{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[v] \neq 0 \qquad \langle P, \texttt{pc} + 1, \Sigma \rangle \to \Sigma'}{\langle P, \texttt{pc}, \Sigma, \Theta \rangle \to \Sigma'}$$

$$[\textsc{BnzSem}]\quad \frac{P[\texttt{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[v] = 0 \qquad \langle P, l, \Sigma \rangle \to \Sigma'}{\langle P, \texttt{pc}, \Sigma \rangle \to \Sigma'}$$

(a) (3 Pontos) Escreva a semântica natural da instrução $\texttt{mov}(v_1, v_2)$, que copia o conteúdo da variável v_2 para a variável v_1. Dica: observe a regra AddSem logo acima.

(b) (3 Pontos) Escreva a semântica natural da instrução $\texttt{ldm}(v_1, v_0)$, que é equivalente à atribuição $v_1 = *v_0$. Isto é, essa instrução lê o conteúdo de v_0, usa esse conteúdo como um endereço na memória Σ, e copia o conteúdo desse endereço para a variável v_1. Dica: observe a regra StmSem logo acima.

(c) (4 Pontos) Escreva a semântica natural da instrução $\texttt{jmp}(l)$, que desvia o contador de programas para o rótulo l. Dica: observe as regras BzrSem e BnzSem logo acima.

3. Essa questão refere-se aos predicados abaixo, escritos em Prolog.

```
@([], L, L).
@([H|L1], L2, [H|X]) :- @(L1, L2, X).

rev([], []).
rev([H|T], X) :- @(XX, [H], X),  rev(T, XX).
```

(a) (2 Pontos) Uma das duas buscas a seguir não termina: (i) `rev([1], Q)` ou (ii) `rev(Q, [1])`. Qual dessas buscas, (i) ou (ii), não termina?

(b) (8 Pontos) Justifique a resposta dada acima mostrando a árvore de buscas da *query* que termina.

4. (10 Pontos) Nessa questão deverá ser desenvolvido em Prolog um sistema experto que encontre pares entre potenciais casais de namorado. Por simplicidade assumiremos que rapazes e moças em nosso mundo podem ter alguns dentre os quatro interesses: *dançar*, *caminhar*, *rir* e *brincar*. O melhor par para uma moça é o rapaz com quem ela tem mais interesses em comum. Uma forma de representar esses interesses é via os predicados abaixo. Nesse exemplo, assumimos que temos cadastrados em nosso sistema quatro moças e quatro rapazes:

```
dance_g(ana, 1).    dance_b(alberto, 0).     track_g(ana, 0).    track_b(alberto, 1).
dance_g(rosa, 0).   dance_b(rafael, 1).      track_g(rosa, 0).   track_b(rafael, 0).
dance_g(julia, 1).   dance_b(mauro, 1).      track_g(julia, 1).   track_b(mauro, 1).
dance_g(teresa, 0).   dance_b(jose, 0).      track_g(teresa, 1).   track_b(jose, 1).

laugh_g(ana, 1).    laugh_b(alberto, 0).     play_g(teresa, 1).    play_b(jose, 0).
laugh_g(rosa, 1).    laugh_b(rafael, 1).     play_g(julia, 1).    play_b(mauro, 1).
laugh_g(julia, 0).    laugh_b(mauro, 1).     play_g(rosa, 1).    play_b(rafael, 0).
laugh_g(teresa, 0).    laugh_b(jose, 0).     play_g(ana, 0).    play_b(alberto, 0).
```

Escreva um predicado `sweetheart(B, G)` que seja verdade quando a moça `G` for o melhor par para o rapaz `B`. O melhor par é definido pela quantidade de interesses em comum. Por exemplo, o melhor par para `jose` é `teresa`, pois ambos *não* gostam de dançar, gostam de caminhar, e *não* gostam de rir. A única divergência entre eles acontece no predicado `play`: a menina, nesse caso, é brincalhona, e o rapaz é sério. Assim, considerando o banco de predicados visto logo acima, teríamos as seguintes buscas:

```
?- sweetheart(jose, X).
X = teresa ;

?- sweetheart(mauro, X).
X = julia ;

?- sweetheart(rafael, X).
X = ana ;

?- sweetheart(alberto, X).
X = teresa ;
```

Note que o seu predicado deve ser geral o suficiente para lidar com outros bancos de verdades, além desse visto como exemplo. Finalmente, em caso de empate, isto é, se houver duas moças com a mesma quantidade de afinidades por um rapaz, seu predicado pode usar qualquer uma delas.

5. (10 Pontos) Várias linguagens utilizam o conceito de *classe* para implementar tipos abstratos de dados. Algumas dessas linguagens, inclusive, provêem uma noção muito flexível de classe. Por exemplo, em Python classes são valores de primeira ordem, que podem ser modificados em tempo de execução por exemplo. Essa capacidade é muito útil, pois é possível estender-se um programa em direções muito diferentes daquela que motivou o projeto original daquele código. Por exemplo, abaixo vemos dois módulos implementados em Python. O primeiro, que define algumas formas geométricas, está no arquivo `shapes.py`. O segundo arquivo, `uses_shapes.py`, usa aquelas formas. Porém, as nossas formas em `shapes.py` não possuem o método `area`. Portanto elas não respeitam o contrato exigido pelo método `get_volume_by_height` em `uses_shapes.py`.

shapes.py

```
class Circle:
  def __init__(self, radius):
    self.radius = radius

class Rectangle:
  def __init__(self, width, height):
    self.width = width
    self.height = height

class Composite:
  def __init__(self):
    self.elements = []
  def add(self, shape):
    self.elements.append(shape)
```

use_shapes.py

```
from shapes import Circle,
Rectangle, Composite

Insira o seu código aqui para que o resto
deste módulo funcione corretamente.

def get_volume_by_height(shape, height):
  return shape.area() * height

c = Circle(4.0)
r = Rectangle(3.0, 5.0)
e = Composite()
e.add(c)
e.add(r)

print get_volume_by_height(c, 2.5)
print get_volume_by_height(r, 2.5)
print get_volume_by_height(e, 2.5)
```

Nessa questão você deverá preencher a lacuna em `uses_shapes.py` para que o método `get_volume_by_height` funcione corretamente. Note que você **não** tem acesso ao código fonte em `shapes.py`.

- A área de um círculo é seu raio vezes π.
- A área de um retângulo é sua largura vezes sua altura.
- A área de uma composição (`composite`) é a soma das áreas de todos os elementos que dela fazem parte.

6. (1 Ponto cada) Considere o programa abaixo, que foi escrito em uma linguagem hipotética, que chamaremos de *rhyme*:

```
def test (int x) {
  a[1] = 6;
  e = 2;
  x += 3;
}
main() {
  a[1] = 1; a[2] = 2; e = 1;
  test(a[e]);
  print a[1], " ", a[2], " ", e);
}
```

Preencha a tabela abaixo com os valores impressos de acordo com as diferentes políticas de passagem de parâmetros:

Elemento	`a[1]`	`a[2]`	`e`
Nome			
Valor			
Referênica			

- (1 Ponto) Dê uma sugestão de como o curso de LP poderia ser melhorado.