Prova Final de Linguagens de Programação
- DCC024B -
Sistemas de Informação

Ao concordar participar da prova, você dá sua palavra de honra que suas respostas são fruto único do seu trabalho. Você pode consultar a internet, por exemplo, mas não pode consultar outros seres vivos para fazer a prova.

As regras do jogo:

- Você pode consultar entidades inanimadas, mas não pode consultar entidades animadas.
- Você precisa enviar três arquivos para `fernando@dcc.ufmg.br` até as 19h00 do dia 28 de Novembro, a saber:
  - `sol.sml`: solução da questão 1.
  - `sol.py`: solução da questão 2.
  - `sol.pl`: solução da questão 3.
- Seu e-mail deve ter o título `Prova_DCC024`. Não use qualquer outro título.
- O único texto no corpo de seu e-mail deve ser seu nome, e, possivelmente, a resposta da questão extra. Não escreva o código dos arquivos no e-mail: envie os arquivos como anexos.
- Caso você queira reenviar suas respostas, simplesmente responda a mensagem de seu e-mail anterior. Não envie e-mails separados!
- Não há como tirar dúvidas durante a prova. Lembre-se da cláusula sobre entidades animadas. Isso inclui escrever e-mails para o instrutor.
- Seja honesto e lembre-se: **você deu sua palavra de honra**.

Tabela 1: Pontos acumulados (para uso do instrutor)

Questão 1	Questão 2	Questão 3	Extra 0.5

Questão extra (0.5): se o H era um herói, e o V um vilão, o que fazia o F para viver?

1. A questão abaixo diz respeito à seguinte definição de expressões aritméticas, implementada como um tipo algébrico em SML:

```
datatype Exp = ADD of Exp * Exp | MUL of Exp * Exp | NUM of int
```

   (a) (5 Pontos) Escreva uma função `eval` em SML, cujo tipo seja `Exp -> int`. Essa função transforma uma expressão aritmética em um inteiro em SML:

```
- eval (ADD (NUM 4, NUM 5));
val it = 9 : int

- eval (MUL ((ADD (NUM 4, NUM 5)), MUL (NUM 3, NUM 6)));
val it = 162 : int
```

   (b) (5 Pontos) Implemente uma função `mul2add`, cujo tipo é: `Exp -> Exp`. Essa função transforma uma multiplicação em uma sequência de adições. Em outras palavras, ela transforma a expressão $e_1 \times e_2$ em e_2 somas: $e_1 + e_1 + \ldots + e_1$. Por exemplo:

```
- mul2add(MUL(NUM 3, NUM 4));
val it = ADD (NUM 4, ADD (NUM 4, ADD (NUM 4, NUM 0))) : Exp

- mul2add(MUL(NUM 2, NUM 4));
val it = ADD (NUM 4, ADD (NUM 4, NUM 0)) : Exp

(* Caso ela receba expressões que não são multiplicação, então nada
   acontece *)
- mul2add (ADD (NUM 3, NUM 4));
val it = ADD (NUM 3,NUM 4) : Exp
```

   Note que, dependendo do interpretador que você usar, sua saída pode ser um pouco diferente. Isto não está errado. Por exemplo:

```
- mul2add(MUL(NUM 3, NUM 4));
val it = ADD (NUM 4,ADD (NUM #,ADD #)) : Exp

- mul2add(MUL(NUM 2, NUM 4));
val it = ADD (NUM 4,ADD (NUM #,NUM #)) : Exp (* Note a # *)
```

2. Nesta questão você deverá utilizar a mesma definição de expressões vista anteriormente, mas desta vez, nós a faremos em Python, a partir das seguintes classes[1]:

```
class Num:
  def __init__(self, n):
    self.num = n

class Add:
  def __init__(self, e1, e2):
    self.e1 = e1
    self.e2 = e2

class Mul:
  def __init__(self, e1, e2):
    self.e1 = e1
    self.e2 = e2
```

(a) (3 Pontos) Adicione um método `__str__(self)` às três classes acima, para que você possa converter instâncias de expressões em `String`. O método `__str(self)__` é o equivalente a `toString` em Java. Por exemplo:

```
>>> execfile('sol.py')
>>> n0 = Mul(Add(Num(2), Num(3)), Num(4))
>>> print n0
2 + 3 * 4
>>> n1 = Add(n0, n0)
>>> print n1
2 + 3 * 4 + 2 + 3 * 4
```

Sua implementação deve deixar um espaço entre operadores e operandos. Não deve haver espaços no início e no final da expressão.

(b) (3 Pontos) Implemente uma função `eval(e)`, que receba uma expressão `e`, e retorne o seu valor. Por exemplo:

```
>>> execfile('sol.py')
>>> n0 = Mul(Add(Num(2), Num(3)), Num(4))
>>> eval(n0)
20
>>> n1 = Add(n0, n0)
>>> eval(n1)
40
```

(c) (4 Pontos) Adicione uma classe `Square` à seu conjunto de expressões. Uma instância de `Square` representa o quadrado de uma expressão. Por exemplo:

```
>>> execfile('sol.py')
>>> n0 = Add(Square(Num(2)), Square(Num(3)))
>>> eval(n0)
13
>>> print n0
(2)^2 + (3)^2
>>> n1 = Square(Num(5))
```

[1] Fique à vontade para adicionar quaisquer métodos auxiliares às classes `Zero` e `Succ`.

```
>>> print n1
(5)^2
>>> eval(n1)
25
>>> n2 = Square(n0)
>>> print n2
((2)^2 + (3)^2)^2
```

Sua implementação de `Square` precisa ter, no mínimo, os métodos `__init__` e `__str__`.

3. A solução desta questão deve ser implementada em Prolog.

   (a) (3 Pontos) Implemente um predicado `range(M, N, L)`, que seja verdade se `L` for a lista com todos os inteiros de `M` até `N` inclusive. Por exemplo:

```
?- range(1, 1, L).
L = [1] ;
false.

?- range(1, 9, L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
false.

?- range(10, 9, L).
false.
```

   (b) (2 Pontos) Implemente o predicado `triple_prod(N, X, Y, Z)`, que seja verdade se `N = X * Y * Z`. Por exemplo:

```
?- triple_prod(24, 2, 3, 4).
true.

?- triple_prod(N, 2, 3, 4).
N = 24.
```

   (c) (3 Pontos) Implemente o predicado `seq_trip_prod(N, X, Y, Z)`, que seja verdade se:

   - `triple_prod(N, X, Y, Z)` for verdade;
   - `X < Y` e `Y < Z` for verdade.

   Por exemplo:

```
?- seq_trip_prod(60, X, Y, Z).
X = 2,
Y = 3,
Z = 10 ;
X = 2,
Y = 5,
Z = 6 ;
X = 3,
Y = 4,
Z = 5 ;
false.
```

   (d) (2 Pontos) Implemente o predicado `all_triples(N, Triples)`, que seja verdade se `Triples` for uma lista com todas as triplas `(X, Y, Z)` tal que `seq_trip_prod(N, X, Y, Z)` seja verdade. Por exemplo:

```
?- all_triples(60, Triples).
Triples = [ (2, 3, 10), (2, 5, 6), (3, 4, 5)].

?- all_triples(120, Triples).
Triples = [ (2, 3, 20), (2, 4, 15), (2, 5, 12), (2, 6, 10), (3, 4, 10),
            (3, 5, 8), (4, 5, 6)].

?- all_triples(200, Triples).
Triples = [ (2, 4, 25), (2, 5, 20), (4, 5, 10)].
```