Faculdade de Computação
Programação Funcional (BSI) – 1° Período
1a. Avaliação - Questões e Gabarito - (20 pontos)

1) Para as expressões abaixo, escritas em Haskell, descreva a operação realizada e informe o resultado obtido (3 pontos):

```
> length ("haskell" ++ ".hs")
```

A função `length` retorna o tamanho de uma lista, neste caso a função retorna o tamanho da lista resultante da concatenação das duas listas de caracteres: `"haskell"` e `".hs"`. O símbolo (++) é o operador de concatenação. O resultado é 10.

```
> if 12>5 then 100 else 200
```

A instrução **if** define uma seleção bidirecional, o que permite escolher entre dois caminhos de execução, tendo em consideração o resultado de uma expressão. Neste caso a expressão (12>5) é verdadeira, então o resultado é 100.

```
> ( 3 + 4 - 5 ) == ( 3 - 5 + 4 )
```

A expressão acima indica uma comparação entre duas expressões aritméticas. Se ambas retornarem o mesmo valor, são consideradas iguais. O resultado é verdadeiro (True).

```
> fn 5 3 where fn a b = a + b^2
```

A expressão representa a aplicação da função **fn** aos valores 5 e 3 sendo que fn é definida localmente para dois argumentos, retornando a soma de ambos sendo o segundo elevado ao quadrado. O resultado é 14.

```
> fnp (fnp 4 1 4) 1 6 where fnp a b c = a * (b ** c)
```

Neste caso, uma função denominada fnp é aplicada duas vezes para 3 valores inteiros e é definida localmente multiplicando-se o primeiro pelo resultado do segundo elevado à potência do terceiro. O resultado é 4.0 (tipo Float, pois o operador `**` retorna um valor do tipo Float) .

```
> let cubo n = n^3 in cubo 5
```

A expressão **let** permite definir um bloco no qual podem ser feitas definições locais. Neste caso foi feita uma definição local para a função cubo, que será aplicada ao valor 5. O resultado é 125.

```
> (+) (gcd 24 6) (length "ana")
```

A expressão acima é a soma do máximo divisor comum entre 24 e 6 (resultado 6) com o tamanho da lista de caracteres “ana” (resultado 3). O resultado final é 9.

```
> (10,20,30) < (1,2,3)
```

A expressão acima compara duas tuplas, verificando se os elementos da primeira são menores que os elementos da segunda. O resultado é falso (`False`).

```
> let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

As expressões let definem localmente valores para as variáveis x y e z, que serão utilizadas para a realização de um cálculo. O resultado é 9.

2) Escreva uma função `sinal::Int -> Char` em Haskell que recebe um número inteiro e retorna (3 pontos):

- ’+’ se o número estiver entre 1 e 9
- ’0’ se o número for 0,
- ’-’ se o número estiver entre −1 e −9
- senão, indica um erro

Resposta:

```
sinal::Int->Char
sinal n |(n >= 1) && (n <= 9) = '+'
        |(n == 0) = '0'
        |(n <= -1) && (n >= -9) = '-'
        |otherwise = error "Nao ha resposta"
```

3) Seja a função `primo` definida para verificar se um determinado número inteiro é primo (divisível apenas por si mesmo, e por 1). Explique a função `primo` e suas funções auxiliares e forneça o resultado, mostrando passo-a-passo a execução da chamada com `n=31`. (4 pontos)

```
primo n | n < 1 = error "nao eh um numero positivo"
        | n == 1 = False
        | otherwise = verifPrimo n == n

verifPrimo n = verifPrimoIni n 2

verifPrimoIni n k | divide n k = k
        | k^2 > n = n
        | otherwise = verifPrimoIni n (k+1)

divide n d = mod n d == 0

> primo 31
```

Resposta:

A função primo recebe um número *n* e verifica se o mesmo é negativo ou igual a 1 e trata inicialmente estes dois casos. Senão a função primo testa se uma outra função (verifPrimo) aplicada ao número retorna o próprio número. Se isso acontecer, o número é primo.

Por sua vez, a função verifPrimo utiliza uma nova função (verifPrimoIni) que analisa o número e inicia uma variável *k* com o valor 2. O primeiro teste (`divide n k`) indica que se o resto da divisão do número por *k* for 0, então o valor de *k* deve ser retornado. Neste caso encontramos um divisor para *n*.

Exemplo: `verifPrimoIni 2 2` retorna 2, e dois deverá ser avaliado como primo. Já `verifPrimoIni 4 2` retorna 2, e 4 não será avaliado como primo.

Caso o resto da divisão não seja zero, continuaremos a buscar um divisor para o número *n*, até que *k*^2 seja maior que *n*. Neste caso, a função não encontra um divisor, e retorna o próprio número.

```
>>>>>>> Execução passo a passo: <<<<<<

primo 31   = verifPrimo 31 == 31 =
           = 31 == 31
           = True

verifPrimo 31 = verifPrimoIni 31 2 = 31


verifPrimoIni 31 2 =
     verifPrimoIni 31 3 =
          verifPrimoIni 31 4 =
               verifPrimoIni 31 5 =
                    verifPrimoIni 31 6 = 31
```

```
Main> primo 31
True
```

4) Números reais podem ser utilizados para representar quantias em dinheiro, e descontos em porcentagem. Quantidades podem ser representadas por valores inteiros:

```
type Moeda = Float
type Porcent = Float
type Quantidade = Int
```

Escreva uma função `desconto::Quantidade -> Porcent` para implementar as regras:

- Para quantidade entre 1 e 9, não há desconto.
- Para quantidade entre 10 e 40, existe desconto de 10%.
- Para quantidade 50 ou superior, o desconto será de 20%.

Resposta:

```
desconto::Quantidade -> Porcent
desconto n | (n>=1) && (n<=9) = 1
         | (n>=10) && (n<=40) = 0.9
           | (n>=50) = 0.8
           | otherwise = error "valor negativo ou zero"
```

5) Faça um algoritmo que calcule o salário e o imposto de renda de um grupo de pessoas, considerando:

a) Para cada pessoa tem-se: Nome, Número de Dependentes, Renda Anual;

```
type Nome = String
type Dependentes = Int
type Renda = Float
type Pessoa = (Nome, Dependentes, Renda)
```

b) O imposto é calculado segundo a tabela abaixo:

Renda:	
até R$ 10.000,00	-->isento
--> R$ 10.000,00 até R$ 30.000,00	--> 5% do valor
--> R$ 30.000,00 até R$ 60.000,00	--> 10%
--> R$ 60.000,00	--> 15%

c) Há um desconto de R$ 600,00 (no imposto) para cada dependente.

```
salario::Pessoa->Float
salario (n,d,r) = r / 12

> salario ("ana", 2, 30000)
2500.0
```

```
imposto::Pessoa->Float
imposto (n,d,r)
  | r >= 0 && r <= 10000 = 0
  | r > 10000 && r <= 30000 = r*0.05 - (600 * (fromIntegral d))
  | r > 30000 && r <= 60000 = r*0.1 - (600 * (fromIntegral d))
  | r > 60000  = r*0.15 - (600 * (fromIntegral d))
  | otherwise = error "valor negativo"


Main> imposto ("ana", 2, 30000)
300.0
Main> imposto ("ana", 2, 60000)
4800.0
Main> imposto ("ana", 8, 60000)
1200.0
Main> imposto ("ana", 8, 15000)
-4050.0

-- Para evitar um imposto negativo, podemos definir a função:
calcImposto (n,d,r) = if imposto (n,d,r) < 0 then 0
                                             else imposto (n,d,r)

Main> calcImposto ("ana", 8, 15000)
0.0
```

(Esta questão não precisa ser resolvida em linguagem Haskell. As funções podem ser apresentadas na forma de algoritmos, como informa o enunciado.)