UFU/FACOM
Disciplina: Programação Funcional
Período: 2018/1
Ref: Terceiro Trabalho de Programação
Data de Entrega:

1 - Título: Simulação de Filas

2 - Objetivo: projetar um programa que permita simular filas. Os seguintes tópicos estarão sendo explorados neste trabalho: listas infinitas, tipos algébricos de dados, tipos sinônimos, listas, filas, geração de números aleatórios e avaliação preguiçosa.

3 - Descrição: suponha que desejemos modelar, ou simular, como as filas num banco (ou em outro local) se comportam. Podemos usar este estudo para decidir quantos caixas necessitariam estar trabalhando em determinadas horas do dia. O sistema utilizará como dado de entrada as chegadas dos clientes e como dado de saída as saídas dos clientes. Cada um destes dados pode ser modelado por um tipo.

- `ClienteQChega` será usado como tipo do cliente que chega ao banco. Em um dado momento existem duas possibilidades:
  - Nenhum cliente chega. Isto será representado pelo construtor `Nao`.
  - Algum cliente chega. Isto será representado pelo construtor `Sim`. Este construtor terá como componentes o horário de chegada do cliente e o tempo despendido para atendê-lo.

```
data ClienteQChega = Nao | Sim TempoQChegou TempoPAtend
type TempoQChegou = Int
type  TempoPAtend = Int
```

- `ClienteQSai` será usado como tipo da mensagem da saída. Ou ninguém deixa o local, representado pelo construtor `Nenhum` ou uma pessoa é liberada, o que é representado pelo construtor `Liberado` que tem como componentes o tempo que o cliente chegou, o tempo de espera para ser atendido e o tempo despendido para atendê-lo. Desta forma temos:

```
data ClienteQSai = Nenhum
          |Liberado TempoQChegou TempoDEsp TempoPAtend
type TempoDEsp = Int
```

Para implementar o sistema será necessário definir um tipo de dado que represente uma fila. Na verdade este tipo é bastante conhecido em computação. Sua definição e as funções que o utilizam são definidas a seguir.

```
type Fila  = [Int]
```

A fila é constituída por uma coleção de objetos. Neste caso estamos representando esta coleção como uma lista de inteiros. As operações mostradas a seguir estão relacionadas a listas.

```
estaVazia :: Fila -> Bool
estaVazia [] = True
estaVazia _ = False

enfileira :: Int ->  Fila -> Fila
enfileira a x =  x ++ [a]

desenfileira :: Fila -> (Int,Fila)
desenfileira x
     | not (estaVazia x)      = (head x,tail x)
     | otherwise              = error "erro: a fila esta
                                       vazia"
```

A operação `estaVazia` verifica se a fila está vazia. A operação `enfileira` coloca um elemento na fila que passa a ser o ultimo da fila. A operação `desenfileira` remove um elemento da fila, que no caso é o primeiro elemento da fila. Existe uma implementação alternativa para `enfileira` e `desenfileira`. Observe:

```
enfileira a x = (a:x)
desenfileira x
     | not (estaVazia x) = (last x,init x)
     | otherwise         = error
                           "erro: a lista esta vazia"
```

Qual destas duas implementações você considera mais eficiente?

Existe uma forma eficiente de tratar com as duas operações. A idéia consiste em usar duas listas de tal forma que adicionar ou remover um elemento pode acontecer na cabeça da lista. Observe a seguir o código que define a fila por meio de duas listas.

```
estaVazia  ([],[]) = True
estaVazia _        = False

enfileira a (l,r) = (l,(a:r))

desenfileira ((a:l),r) = (a,(l,r))
desenfileira ([],[])    = error "erro: fila vazia"
desenfileira ([],r) = desenfileira (reverse r,[])
```

Depois de definido o tipo `Fila` e suas operações relacionadas retornemos ao problema da simulação. Para facilidade de entendimento suponha que o tempo do sistema é

medido em minutos. O tipo `ClienteQChega` associado a `Nao` indica que nenhum cliente chegou ao passo que `Sim 30 10` especifica a chegada de um cliente no tempo 30 e precisará de 10 minutos para ser atendido. O tipo `ClienteQSai` associado a `Liberado 10 20 5` especifica que o cliente chegou no tempo 10, esperou 20 minutos antes de ser atendido e foi atendido em 5 minutos.

Inicialmente vamos simular o processo de enfileiramento. Para isto vamos criar um tipo `EstadoDaFila`. Existem duas operações principais associadas a este tipo. A primeira é adicionar um novo item, um `ClienteQChega`, para a fila. A segunda é processar a fila a cada minuto, o que consiste em avaliar o item que está na cabeça da fila. Duas saídas são possíveis: o item pode ter seu processamento concluído tal que um `ClienteQSai` é gerado ou continuar no processo. Outros elementos necessários incluem uma fila vazia, uma indicação do comprimento da fila e um teste para saber se a fila está vazia.

A descrição feita anteriormente permite descrever as assinaturas das operações que definirão o tipo `EstadoDaFila`.

```
adicionaCliente:: ClienteQChega -> EstadoDaFila -> EstadoDaFila
processaFila :: EstadoDaFila -> (EstadoDaFila, [ClienteQSai])
tamanhoDaFila :: EstadoDaFila -> Int
filaVazia :: EstadoDaFila -> Bool
```

O tipo `EstadoDaFila` permite-nos modelar uma situação na qual os clientes são atendidos por único processador (ou caixa, se pensarmos em um banco). Como podemos modelar o caso onde exista mais que uma fila? Vamos modelar esta situação através de um tipo de dado `EstadoDoServidor`, que chamaremos de *servidor*.

Um servidor consiste de uma coleção de filas que podem ser identificadas pelos inteiros 0, 1, e assim por diante. Vamos assumir que o sistema recebe um `ClienteQChega` a cada minuto: uma pessoa no máximo pode chegar a cada minuto.

Existem três operações principais relacionadas a um servidor. A primeira deve ser capaz de adicionar um `ClienteQChega` a uma das filas. A segunda permite processar cada elemento das filas por um período de tempo: isto pode gerar uma lista de `ClienteQSai` pois cada fila pode gerar tal mensagem. A terceira, através de um passo da simulação, permite combinar um passo do servidor com a alocação do `ClienteQChega` na menor fila no `EstadoDoServidor`.

Existem três operações adicionais. A primeira permite iniciar um servidor com um número apropriado de filas vazias. A segunda possibilita identificar o numero de filas num servidor. A terceira fornece a menor fila existente no servidor.

Em função da descrição anterior teremos as seguintes assinaturas para as operações:

```
colocaNumaFila :: Int -> ClienteQChega -> EstadoDoServidor
                -> EstadoDoServidor
```

```
processaServidor :: EstadoDoServidor
                -> (EstadoDoServidor, [ClienteQSai])
processaSimulacao :: EstadoDoServidor -> ClienteQChega
                -> (EstadoDoServidor,[ClienteQSai])
tamanhoDoServidor :: EstadoDoServidor -> Int
menorFila :: EstadoDoServidor -> Int
```

Implementando a simulação

Após especificadas as assinaturas das funções associadas a `EstadoDaFila` e `EstadoDoServidor` vamos agora implementar as funções. Inicialmente começando com `EstadoDaFila` temos que:

- Deve existir uma fila de `ClienteQChega` a ser processada. Ela pode ser representada por uma lista e podemos pegar a cabeça desta lista como o item atualmente sendo processado;
- É necessário manter um registro do tempo de processamento alocado para o item da cabeça até o tempo particular representado pelo estado;
- Em um `ClienteQSai`, é preciso fornecer o tempo de espera para o item particular sendo processado. Nós sabemos o tempo de chegada e o tempo necessário de processamento. Se soubermos o tempo atual podemos calcular o tempo de espera.

Portanto uma definição para `EstadoDaFila` poderia ser:

```
type EstadoDaFila = (Tempo,TempoDeAtend,[ClienteQChega])
```

onde o primeiro campo fornece o tempo atual, o segundo item fornece o tempo de atendimento até o presente momento para o item sendo processado e o terceiro a fila. Examinemos agora, uma a uma, as operações.

Novos clientes são colocados no fim da lista de clientes. Desta forma:

```
adicionaCliente :: ClienteQChega -> EstadoDaFila -> EstadoDaFila
adicionaCliente m (tempo,tempDeAtend,ml) =
                            (tempo,tempDeAtend, ml++[m])
```

A definição mais complicada é da operação `processaFila`. Como já foi explicado informalmente existem dois casos principais quando um item está sendo processado:

```
processaFila ::EstadoDaFila -> (EstadoDaFila,[ClienteQSai])
processaFila (tempo,tempDeAtend,(Sim a tempNecDAtend:resto))
 | tempDeAtend < tempNecDAtend =
     ((tempo+1),tempDeAtend+1,
      ((Sim a tempNecDAtend:resto),[])
     )
 | otherwise       =
     ((tempo+1,0,resto),
      [Liberado a (tempo-tempNecDAtend-a) tempNecDAtend]
     )
```

No primeiro caso, quando o tempo de serviço prestado (`tempDeAtend`) é menor que o necessário (`tempNecDAtend`) o processamento não acaba. Portanto, adiciona-se 1 a `tempo` e `tempDeAtend` e nenhum cliente é liberado.

No segundo caso o processamento é concluído. O novo estado da fila é `(tempo+1,0,resto)`. Quando um cliente é liberado tem-se que o tempo despendido para atendê-lo é dado por subtrair o tempo necessário para atendimento e o tempo de chegada do tempo atual `(tempo-tempNecDAtend-a)`.

Se não existe nada a ser processado basta avançar o tempo atual por 1 e não produzir nenhuma saída.

```
processaFila (tempo,tempDeAtend,[]) =
                ((tempo+1,tempDeAtend,[]),[])
```

Observe que um cliente `Nao` não é tratado, pois estes clientes são filtrados pelo servidor. As funções restantes são especificadas a seguir.

```
filaDeInicio :: EstadodaFila
filaDeInicio = (0,0,[])

tamanhoDaFila :: EstadoDaFila -> Int
tamanhoDaFila (tempo,tempoDeAtend,l) = length l

filaVazia :: EstadoDaFila -> Bool
filaVazia (t,s,q) = (q == [])
```

Obviamente existem outras implementações possíveis.

O servidor consiste de uma coleção de filas acessadas por inteiros começando a partir de 0. Podemos representar o seu tipo por uma lista de filas. Assim temos que:

```
type EstadoDoServidor = [EstadoDaFila]
```

A inclusão de um elemento numa fila, realizada pela função `colocaNaFila`, utiliza a função `adicionaCliente` descrita anteriormente.

```
colocaNaFila :: Int -> ClienteQChega -> EstadoDoServidor
                    -> EstadoDoServidor
colocaNaFila n im st
     = take n st
       ++ [novoEstadoDaFila]
       ++ drop (n+1) st
       where
          novoEstadoDaFila = adicionaCliente im (st!!n)
```

O processamento do **servidor** é feito por avaliar cada fila a ele pertencente e concatenar as saídas que elas produzem.

```
processaServidor :: EstadoDoServidor
                 -> (EstadoDoServidor, [ClienteQSai])
processaServidor [] = ([],[])
processaServidor (q:qs) = ((nq:nqs), mess ++ messes)
where
     (nq,mess)    = processaFila q
     (nqs,messes) = processaServidor qs
```

Um passo de processamento da simulação consiste em processar o servidor e então adicionar um cliente, caso chegue, à menor fila.

```
processaSimulacao :: EstadoDoServidor -> ClienteQChega
                  -> (EstadoDoServidor,[ClienteQSai])
processaSimulacao estServ im =
     (adicionaNovoObjeto im estServ1,clientQSai)
     where
          (estServ1,clientQSai) = processaServidor estServ
```

A colocação de um cliente na menor fila é feita pela função `adicionaNovoObjeto`, que não está na assinatura descrita anteriormente. A razão para isto é que ela pode ser definida por meio das funções `colocaNaFila` e `menorFila`.

```
adicionaNovoObjeto :: ClienteQChega -> EstadoDoServidor
                                    -> EstadoDoServidor
adicionaNovoObjeto Nao estServ  = estServ
adicionaNovoObjeto (Sim tempoDeChegada tempoNecAtend) estServ
     = colocaNaFila (menorFila estServ)
                (Sim tempoDeChegada tempoNecAtend) estServ
```

Observe que é nesta função que os clientes `Nao` que chegam não são passados para as filas.

As outras funções de interesse aparecem especificadas a seguir.

```
estadoInicialDoServidor :: EstadoDoServidor
estadoInicialDoServidor = copy nroDeFilas filaDeInicio
```

A função `copy` faz com que o servidor seja iniciado com determinado número de filas (`nroDeFilas`) sendo que cada fila é inicialmente dada por `filaDeInicio`.

O número de filas presente num servidor é dado pela função `tamanhoDoServidor` que é basicamente a função `length`.

```
tamanhoDoServidor :: EstadoDoServidor -> Int
tamanhoDoServidor = length
```

Na obtenção da menor fila pode-se utilizar a função `tamanhoDaFila` do tipo `EstadoDaFila`.

```
menorFila :: EstadoDoServidor -> Int
menorFila [q] = 0
menorFila (q:qs)
  | tamanhoDaFila (qs!!menor) <= tamanhoDaFila q = menor + 1
  | otherwise                                    = 0
  where
     menor = menorFila qs
```

Assim concluímos a implementação das operações relacionadas aos dois principais tipos que serão utilizados: `EstadoDaFila` e `EstadoDoServidor`.

Neste processo de simulação existe a necessidade de se trabalhar com números aleatórios. Antes de qualquer coisa, deve-se observar que nenhum programa Haskell pode produzir uma sequência verdadeiramente aleatória de números. Afinal de contas queremos sempre ser capazes de prever o comportamento de programas e aleatoriedade é algo imprevisível. Entretanto o que pode ser feito é a geração de uma sequência pseudoaleatória de números naturais menores que um valor que será identificado como `modulo`. Para isto se pode usar o método das **congruências lineares**. O método de congruências lineares inicia com uma semente (valor inicial da sequência de números) e então obtém o próximo valor da sequência usando o elemento que o antecede.

```
proxNumAleat :: Int -> Int
proxNumAleat n = (multiplicador*n + incremento) rem modulo
```

A sequência pseudoaleatória é obtida por

```
seqAleatoria :: (Int -> [Int])
seqAleatoria = iterate proxNumAleat
```

Explique: a remoção dos parênteses colocados no tipo da função `seqAleatoria` altera a definição da função?

Considerando os valores

```
semente = 17489
multiplicador = 25173
incremento = 13849
modulo = 65536
```

A sequência produzida pela chamada de `seqAleatoria semente` gera

[17489, 59134, ...]

Os números nesta sequência variam de 0 a **6535**, e todos ocorrem com a mesma frequência. E se quisermos gerar valores que estejam numa faixa de a até b inclusive? É necessário criar uma escala para a sequência, o que é obtido por meio de um `map`.

```
escalaSequencia :: Int -> Int -> ([Int] -> [Int])
escalaSequencia a b
     = map scala
     where
          scala n = n/denom + a
          faixa = b - a + 1
          denom = modulo/faixa
```

A faixa de números que vai de 0 até `modulo-1` é dividida em `faixa` de blocos do mesmo tamanho. O número `a` é assinalado para valores no primeiro bloco, `a+1` é assinalado para valores no próximo e assim por diante.

No processo de simulação nós queremos gerar para um cliente o tempo que ele necessita para ser atendido. Suponha que o tempo necessário varie de 1 a 6 minutos. Entretanto admita que eles possam ocorrer com diferentes probabilidades. Admita os seguintes valores que definem uma distribuição de probabilidades:

Tempo Para Atendimento	1	2	3	4	5	6
Probabilidade	0.2	0.25	0.25	0.15	0.1	0.05

Necessitamos de uma função que torne tal distribuição de probabilidades num transformador de listas infinitas. Desde que tenhamos uma função transformando valores individuais podemos aplicá-la ao longo da lista.

Podemos representar uma distribuição de objetos do tipo `t` por uma lista de tipo `[(t,Float)]`, onde assumimos que as entradas numéricas são adicionadas de 1. Nossa função transformando valores individuais terá como tipo:

```
geraFuncao :: [(t,Float)] -> (Float -> t)
```

Os números na faixa de 0 a 65535 são transformados em itens de tipo t. A ideia da função é gerar as seguintes faixas para as entradas da lista acima.

Tempo para Atendimento	1	2	3	...
Inicio	0	(m*0.2)+1	(m*0.45)+1	...
Fim	(m*0.2)	(m*0.45)	(m*0.7)	...

Onde m é usado para `modulo`. A definição segue:

```
geraFuncao = geraFun dist 0.0

geraFun ((ob,p):dist) nUlt aleat
     | nProx >= aleat && aleat > nUlt
          = ob
     |otherwise
          = geraFun dist nProx aleat
             where
               nProx = p* fromInteger modulo + nUlt
```

A função `geraFun` tem um argumento extra que transporta a posição na faixa 0 até `modulo - 1` atingida até agora na busca. Ele é inicialmente zero. A função `fromInteger` é usada para converter um `Integer` para um `Float`.

A transformação de uma lista de números aleatórios é dada por:

```
     map (geraFuncao . fromInteger)
```

A distribuição aleatória de tempos que desejamos começa portanto com

```
map (geraFuncao . fromInteger) (seqAleatoria semente)
= [2,5,1,4,3,1,2,…
```

com 6 aparecendo na 35ª posição.

Agora estamos numa posição de integrar todos os conceitos descritos anteriormente. Como mencionado anteriormente, a simulação corresponde a uma função de uma série de clientes de entrada para uma série de clientes de saída. Assim:

```
     simule :: EstadoDoServidor [ClienteQChega]
                          -> [ClienteQSai]
```

onde o primeiro parâmetro é o estado do servidor no início da execução. Anteriormente apresentamos a função que realiza um passo da simulação:

```
processaSimulacao :: EstadoDoServidor ClienteQChega
     -> (EstadoDoServidor, [ClienteQSai])
```

Esta função a partir do estado do servidor e do cliente chegando no corrente minuto retorna o estado após o minuto processado juntamente com o resultado decorrente do processamento das filas naquele minuto (potencialmente toda fila poderia tanto liberar um cliente naquele instante bem como nenhum cliente).

A saída da simulação será dada pelos clientes de saída gerados num primeiro minuto e após isto por aqueles resultados de uma nova simulação iniciando com o estado atualizado:

```
simule estDoServ (im:messes)
     = outmesses ++ simule proxEstDoServ messes
     where
     (proxEstDoServ,outmesses) =
          processaSimulacao estDoServ im
```

Como podemos gerar uma seqüência de entrada? Anteriormente vimos a seqüência de tempos dadas por

```
seqDeTempos
     = map (geraFuncao . fromInteger)
                (seqAleatoria semente)
     = [2,5,1,4, …
```

Estamos admitindo a chegada de uma pessoa por minuto. Desta forma as mensagens de entrada a serem geradas são

```
entradaDaSimulacao = zipWith Sim [ 1..] seqDeTempos
= [Sim 1 2, Sim 2 5, Sim 3 1, …
```

A função `zipWith` é dada por:

```
zipWith f (a:x) (b:y) = ((f a b) : zipWith f x y)
zipWith _ _ _       = []
```

Quais são as saídas produzidas quando executamos a simulação com esta entrada para quatro filas por fazer `numQueues` igual a 4? A saída começa ...

```
simule estadoInicialDoServidor entradaDaSimulacao
= [Liberado 1 0 2, Liberado 3 0 1, Liberado  6 0 1,
     Liberado 2 0 5, Liberado 5 0 3, Liberado 4 0 4,
     Liberado 7 2 2, …
```

As primeiras seis saídas são processadas sem demora, mas a sétima requer um tempo de espera de 2 antes de ser atendida.

Um número infinito de chegadas representadas por `entradaDaSimulacao` obviamente irá gerar um número infinito correspondente de saídas. Podemos obter uma aproximação finita por fornecer a entrada

```
entradaDaSimulacao2 =
     take 50 entradaDaSimulacao  ++ naos
naos = (Nao:naos)
```

onde após uma chegada em cada dos primeiros cinqüenta minutos ninguém mais aparece. Cinqüenta mensagens de saída serão geradas e definimos esta lista de saídas como

```
take 50
   (simule estadoInicialDoServidor entradaDaSimulacao2)
```

Podemos agora experimentar com dados diferentes tal como a distribuição e o número de filas. O tempo total de espera para uma seqüência finita de `ClienteQSai` é dada por

```
tempoDeEsperaTotal :: ([ClienteQSai] -> Int)
tempoDeEsperaTotal  = sum . map tempoDEsp
where
     tempoDEsp (Liberado _ w _) = w
```

Experimentos a serem realizados:
1 – Obtenha o tempo de espera quando existe somente uma fila (um caixa) para atendimento.
2 – Obtenha o número necessário de caixas para fazer com que o tempo de espera caia para zero.
3 – Repita os itens 1 e 2 usando o método *round-robin*. Por este método a alocação dos elementos nas filas faz-se da seguinte forma: o primeiro item é alocado na fila 0, o segundo item é alocado na fila 1, e assim por diante e inicia-se novamente a alocação na fila 0 após ter-se alocado um elemento para a última fila.
4 - Um projeto mais interessante é modelar uma simples fila com vários caixas. Uma forma de se fazer isto é estender o `EstadoDoServidor` com uma fila extra que alimenta as filas individuais: um elemento deixa a fila do alimentador quando uma das filas pequenas está vazia. Isto deve evitar o tempo de espera desnecessário quando alguém faz uma escolha errada da fila e a simulação deve mostrar uma redução no tempo de espera embora menos que se poderia esperar se os tempos são pequenos.

****