Trabalho 1

Simulação de Caixa Automático

Resumo

Com esta atividade pretende-se explorar a construção de programas interativos usando ações de entrada e saída padrão, tratamento de exceções e funções recursivas com listas.

Será desenvolvida uma aplicação para simular um caixa automático simples, capaz de realizar operações de abertura de conta, consulta de saldo, consulta de extrato, saque e depósito.

Sumário

1 Estrutura da aplicação

Em um editor de texto digite o código fonte da aplicação de acordo com as orientações que se seguem. Use o nome de arquivo `caixa1.hs`.

Tarefa 1

Comece definindo o módulo `Main`, que deve exportar a variável `main`.

Tarefa 2

Anote o tipo de `main`: `IO ()`. Observe o tipo usado: `main` é uma ação de E/S que, quando executada, interage com o mundo e retorna a tupla vazia.

Lembre-se que toda aplicação em Haskell deve ter uma ação de E/S chamada `main`. Esta ação é executada pelo sistema de computação quando a aplicação é executada.

Essa é a forma de interação com o mundo em Haskell. Normalmente não é possível executar uma ação de E/S a não ser que ela faça parte da definição de `Main.main`.

Tarefa 3

Faça uma definição preliminar de `main` de forma que `main` seja formada pela combinação sequencial de várias outras ações de E/S, que, quando executadas:

1. cancela a *bufferização* da saída padrão, usando a função `hSetBuffering`, que deve ser importada do módulo `System.IO`,
2. exibe o título da aplicação: `"Simulacao de caixa automatico"`,
3. exibe a versão do programa, que deverá ser definida em uma variável auxiliar global chamada `versao` do tipo `String` como `"v0.1"`, e
4. retorna a tupla vazia.

Posteriormente a ação `main` será expandida para completar a aplicação.

2 Montando o menu de opções

A aplicação, quando executada, deverá repetidamente:

- exibir um menu de opções (apresentano as possíveis ações ao usuário),
- ler uma opção indicada pelo usuário, e
- executar a ação correspondente à opção escolhida,

até que o usuário escolha terminar.

A forma básica de repetição oferecida nas linguagens funcionais é a *recursividade*. Assim para exibir o menu, ler uma opção e executar a ação correspondente repetidamente precisamos escrever uma definição recursiva.

Tarefa 4

Defina a ação de E/S `menu :: IO ()`. Quando executada, esta ação deverá:

1. Exibir o título do menu (o nome do banco), seguido das opções de ação disponíveis, e de uma mensagem solicitando ao usuário para escolher uma das opções:

```
==============================
Banco Funcional
==============================
Opções:
1. Abertura de conta
2. Saldo
3. Extrato
4. Depósito
5. Saque
6. Fim

Escolha uma opção:
```

2. Ler a opção escolhida pelo usuário, associando-a com a variável `opcao`.
3. Executar a ação correspondente à opção escolhida pelo usuário:
   - para a última opção (`Fim`), exiba uma mensagem agradecendo por usar o banco
   - para as demais opções
     - se a opção for válida, por enquanto apenas exiba uma mensagem dizendo que a operação ainda não foi implementada
     - se a opção for inválida, exiba uma mensagem adequada reportando que a opção é inválida
     - em seguida, executar a ação `menu` novamente (execução recursiva)

   Use uma expressão `case` para analisar os possíveis casos.

Tarefa 5

Acrescente a ação `menu` na sequência de ações que define `main`.

3 Representando os dados

Em sua versão inicial o caixa automático trabalhará apenas com uma conta corrente e não serão implementadas as opções de abertura de conta e consulta de extrato.

A informação necessária sobre a conta portanto se resume no valor do seu saldo, que poderá ser representado por um número em ponto flutuante do tipo `Double`.

As operações de saque e depósito alteram o saldo da conta. Como em Haskell as variáveis não são atualizáveis, precimos de um mecanismo para efetivar estas operações que alteram o saldo da conta.

Um **estado** representa os dados do problema em um dado momento da execução da aplicação. Nesta aplicação precisaremos manter o saldo de uma conta bancária. Logo o estado pode ser representado por um número correspondente ao saldo da conta. O tipo `Estado` será definido como um sinônimo para o tipo `Double`.

Para definir um **tipo sinônimo** em Haskell usamos a declaração `type`. Por exemplo, a seguinte declaração da biblioteca padrão define o tipo `String` como um sinônimo para o tipo `[Char]` das listas de caracteres.

```
type String = [Char]
```

Tarefa 6

Defina o tipo `Estado` como sendo um sinônimo para o tipo `Double`.

As operações associadas ao menu, a princípio, podem produzir um novo estado a partir do estado atual. Por exemplo a operação de saque recebe o saldo atual da conta (estado atual) e atualiza o saldo descontando o valor sacado (novo estado). Podemos então modificar `menu` de forma que `menu` passe a ser uma função do tipo `Estado -> IO Estado`, ou seja, a função `menu` recebe um estado (o estado atual) e resulta em uma ação de E/S que, quando executada, interage com o mundo e retorna um novo estado. Este novo estado irá refletir o resultado da operação correspondente à opção do menu escolhida pelo usuário.

Tarefa 7

Modifique o tipo de `menu` para `Estado -> IO Estado`.

Tarefa 8

Modifique a definição de `menu` de modo que `menu` seja uma função de um argumento representando o estado atual. Como ainda não estamos implementando as operações com a conta corrente (e portanto ainda não há mudanças de estado), a chamada recursiva de `menu` deve usar o mesmo argumento da chamada original (ou seja, o mesmo estado).

```
menu estado = ... menu estado ...
```

Tarefa 9

Modifique a execução da ação `menu` na definição de `main` considerando que agora `menu` é uma função e deve ser aplicada a um estado inicial para obter uma ação de E/S. O estado inicial corresponde ao saldo inicial da conta corrente quando a aplicação é executada: zero (`0.0`).

4 Implementando as operações com a conta corrente

Tarefa 10

Defina a função `saldo` que recebe o estado atual e resulta em uma ação de E/S que, quando executada, exibe o saldo atual, e retorna o próprio estado atual, já que a operação de consulta de saldo não altera o estado.

Tarefa 11

Defina a função `deposito` que recebe o estado atual e resulta em uma ação de E/S que, quando executada, lê o valor a ser depositado, verifica se o valor não é negativo, e em caso afirmativo, retorna o estado com o saldo atualizado pelo depósito. Se o valor for negativo, exibe uma mensagem adequada e retorna o próprio estado, pois neste caso não há alteração.

Tarefa 12

Defina a função `saque` que recebe o estado atual e resulta em uma ação de E/S que, quando executada, lê o valor a ser sacado, verifica se o valor não é negativo, e se há saldo suficiente para o saque e em caso afirmativo, retorna o estado com o saldo atualizado pelo saque. Se o valor for negativo ou não houver saldo suficiente, exibe uma mensagem adequada e retorna o próprio estado, pois neste caso não há alteração.

Tarefa 13

Modifique a definição de `menu` de forma a usar as funções `saldo`, `deposito` e `saque` para realizar as operações correspondentes às opções do menu.

5 Uma função para solicitar e ler um dado

Tarefa 14

Defina a função `prompt :: Read a => String -> IO a` que recebe uma string e resulta em uma ação de E/S que, quando executada, exibe a string (normalmente uma mensagem solicitando um dado) e lê um dado informado pelo usuário, e retorna o valor lido. Use a ação `readLn :: Read a => IO a` a definida no prelúdio.

Tarefa 15

Modifique as definições de `menu`, `deposito` e `saque` para usar a função `prompt` para obter os dados necessários informados pelo usuário.

6 Histórico das operações

Para implementar a consulta de extrato no caixa automático, é necessário manter um registro de todos as operações realizadas pelo usuário com a conta corrente. Logo não será suficiente manter apenas o saldo da conta. Para cada operação será necessário conhecer:

- a descrição da operação realizada
- o valor da operação
- o saldo após a operação

Estes dados podem ser agrupados em um tupla. O histórico (sequência) das operações realizadas pode ser represesntado por uma lista.

O histórico será construído em ordem invertida, isto é, a última operação realizada será a cabeça da lista.

Tarefa 16

O estado passa a ser o histórico das operações realizadas até o momento. Redefina o tipo `Estado` como sendo um tipo lista de triplas cujos elementos são do tipo `String`, `Double` e `Double`, nesta ordem, representando respectivamente a descrição da operação, o valor da operação e o saldo da conta após a operação.

Tarefa 17

Modifique as definições das funções `saldo`, `saque` e `deposito`, considerando a nova definição de `Estado`.

Para acessar o saldo atual da conta basta obter o terceiro componente da primeira entrada do histórico. Use casamento de padrão em uma declaração usando `let` para acessar o saldo quando necessário.

As operações de `saque` e `deposito` deverão obter o saldo atual como descrito, e deverão produzir um novo histórico acrescentando o saque ou depósito ao histórico já existente.

Tarefa 18

Defina a função `extrato :: Estado -> IO Estado` que implementa a operação de consulta de extrato. Todos os movimentos registrados no histórico devem ser exibidos na tela, devidamente formatos em colunas com cabeçalho.

Utilize a função `printf` do módulo `Text.Printf` para fazer a formatação dos dados. Consulte a documentação do módulo (`http://hackage.haskell.org/package/base`) para descobrir como usar a função `printf`.

Tarefa 19

Modifique a definição de `menu` para usar a função `extrato` para exibir o extrato da conta corrente.

7 Carteira de contas

Queremos que o nosso banco tenha vários clientes. Para tanto é necessário que seja possível trabalhar com vários contas correntes. A carteira de contas correntes pode ser representada por uma **lista de associações**.

Uma lista de associações permite representar um mapeamento. Cada elemento da lista é um par. O primeiro componente do par é a chave do mapeamento. O segundo componente do par é valor associado à chave. O prelúdio oferece a função `lookup :: Eq a => a -> [(a, b)] -> Maybe b` para pesquisar uma chave na lista. Se ela não for encontrada, o resultado da função é a constante `Nothing`. Se a chave for encontrada, o resultado é o valor `Just v`, onde `v` o valor associado à chave.

Tarefa 20

Redefina o tipo `Estado` como sendo o tipo das listas de associação com chave do tipo `Int` (representado o número do conta) e com valor associado do tipo usado anteriormente para representar o histórico das operações da conta.

Tarefa 21

Modifique as definições das funções `saldo`, `saque` e `deposito`, considerando a nova definição de `Estado`.

Em cada caso será necessário que o usuário informe o número da conta desejado. Em seguida o número da conta deve ser pesquisado na lista de associações. Se for encontrado, a operação deve ser completada como anteriormente. Se não for encontrado, uma mensagem deve ser exibida informando que a conta não existe.

Use a função `lookup` do prelúdio.

8 Mantendo os dados em arquivo

Para que os dados possam ser preservados entre diferentes execuções do programa será necessário obtê-los de um arquivo no início da execução da aplicação, e gravá-los no arquivo ao final da execução.

Tarefa 22

Redefina `main` para obter os dados de um arquivo texto cujo nome será dado como argumento da linha de comando no início da execução, e gravar os dados no mesmo arquivo ao final da execução.

Use as funções do prelúdio:

- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`
- `read :: Read a => String -> a`
- `show :: Show a => a -> String`