Atividade Avaliativa – RA2

Este trabalho pode ser realizado em **grupos de até 4 alunos**. Grupos com mais de 4 membros terão o trabalho anulado. Leia todo este texto antes de começar e siga o seguinte código de ética: você pode discutir as questões com colegas, professores e amigos, consultar livros da disciplina, bibliotecas virtuais ou físicas, e a Internet em geral, em qualquer idioma. Você pode usar qualquer ferramenta de Inteligência Artificial para dúvidas, mas não para fazer o trabalho no seu lugar. O trabalho é seu e deve ser realizado por você. Indicativo de plágio, ou trabalhos realizados por Inteligência Artificial, resultarão na anulação do trabalho.

Os trabalhos entregues serão avaliados por uma ferramenta de Inteligência Artificial, que verificará a originalidade do texto, a autoria do código e a correção dos códigos apresentados. O trabalho deverá ser entregue por meio de um link para um ambiente virtual (gitHub). Neste ambiente, no arquivo README deve existir um link para um ambiente virtual onde o projeto possa ser executado sem modificações. Serão considerados apenas links para o Online GDB ou para o Repl.it.

1. Objetivo

Praticar conceitos de programação funcional, manipulação de estado e operações de I/O em Haskell para desenvolver um programa que gerencie um sistema de inventário. O sistema funcionará de forma interativa, via terminal, sem a necessidade de interface gráfica, registrando todas as operações em um *log* de auditoria e persistindo o estado atual em disco, garantindo que os dados não sejam perdidos entre as execuções.

2. Descrição do Trabalho

O objetivo é desenvolver um sistema em Haskell capaz de:

1. Interagir com o usuário por meio do terminal (Online GDB ou Repl.it).
2. Coletar comandos do usuário para adicionar, remover ou atualizar itens no inventário.
3. Utilizar funções puras para toda a lógica de negócio (manipulação do inventário), separando-as rigorosamente das operações de I/O.
4. Persistir o estado atual do inventário em um arquivo (Inventario.dat), que será sobrescrito a cada operação bem-sucedida.
5. Registrar cada tentativa de operação (sucesso ou falha) em um arquivo de log (Auditoria.log) no modo "append-only".
6. Carregar o estado anterior (lendo Inventario.dat e Auditoria.log) ao iniciar o programa.
7. Hospedar o código-fonte, os arquivos de dados de exemplo e a documentação em um repositório público no GitHub.

2.1. Módulos Principais: Lógica e Dados

A lógica do sistema será definida por tipos de dados e funções puras em Haskell.

Tipos de Dados: Descrevem o domínio do problema. A base de tipos deve conter, no mínimo:

- **Item**: Um registro contendo *itemID* (*String*), *nome* (*String*), *quantidade* (*Int*) e *categoria* (*String*).
- **Inventario**: Um *Map String Item* (do módulo *Data.Map*) para armazenar os itens.
- **AcaoLog**: Um ADT[1] para o tipo de ação (ex: *Add*, *Remove*, *Update*, *QueryFail*).
- **StatusLog**: Um ADT para o resultado (ex: *Sucesso* | *Falha String*).
- **LogEntry**: Um registro para o *log*, contendo *timestamp* (*UTCTime*), *acao* (*AcaoLog*), *detalhes* (*String*) e *status* (*StatusLog*).

*Nota: Todos os tipos de dados definidos devem derivar **Show** e **Read** para permitir a serialização e desserialização de/para o disco.*

Funções Puras: Descrevem a lógica de negócio do sistema. As funções devem ser capazes de:

- Receber o estado atual do inventário e os parâmetros da operação.
- Retornar o novo estado do inventário e a entrada de *log* correspondente, usando o tipo ***Either*** para sinalizar falhas de lógica (ex: estoque insuficiente, item não encontrado).
- Exemplo de assinatura:
  type ResultadoOperacao = (Inventario, LogEntry)
  addItem :: *UTCTime* -> ... -> *Inventario* -> ***Either*** *String ResultadoOperacao*
  removeItem :: *UTCTime* -> ... -> *Inventario* -> ***Either*** *String ResultadoOperacao*

3. Loop Principal e Persistência (IO)

O "motor" do programa será a função *main* (*main IO ()*), que controlará o fluxo do programa:

- Um ponto de entrada principal (*main*) que inicia a aplicação.
- **Inicialização:** Funções que tentam ler *Inventario.dat* e *Auditoria.log* na inicialização. Devem usar *catch* para lidar com a ausência de arquivos (iniciando com um estado vazio).
- **Loop de Execução:** Funções para ler o comando do usuário, obter o *UTCTime* atual, chamar as funções puras de lógica, e processar o resultado (***Either***).
- **Persistência:** Se a operação for bem-sucedida (*Right*), o *main* deve chamar *writeFile* para *Inventario.dat* e *appendFile* para *Auditoria.log*. Se for uma falha (*Left*), deve apenas chamar *appendFile* para o log.

A interação deve ser `clara` e ocorrer inteiramente pelo terminal (GDB Online ou Repl.it).

[1] ADT: Tipo de dado algébrico. Fizemos o NAT, em sala.

4. Relatórios e Validação

Para validar a lógica do sistema, o programa deve incluir um módulo de análise de *logs* e ser submetido a um conjunto de testes manuais.

- **Análise de Logs:** Devem ser implementadas funções puras (ex: *historicoPorItem*, *logsDeErro*, *itemMaisMovimentado*) que recebem a lista de [*LogEntry*] (lida do disco) e geram relatórios.
- **Comando de Relatório:** O *loop* principal deve aceitar um comando (*report*) que executa as funções de análise sobre os *logs* carregados e exibe os resultados.

4.1. Cenários de Teste e Dados Mínimos

A validação do sistema deve seguir requisitos mínimos de dados e execução de cenários.

- **Dados Mínimos:** O sistema deve ser populado (na primeira execução ou via comandos de adição) com um mínimo de **10 Item distintos** para permitir testes de relatórios e lógica.
- **Cenários de Teste Manuais (Não Automatizados):** O grupo deve executar e documentar os seguintes cenários no README.md:
  1. **Cenário 1: Persistência de Estado (Sucesso)**
     - Iniciar o programa (sem arquivos de dados).
     - Adicionar 3 itens.
     - Fechar o programa.
     - Verificar se os arquivos *Inventario.dat* e *Auditoria.log* foram criados.
     - Reiniciar o programa.
     - Executar um comando de "listar" (a ser criado) ou verificar se o estado carregado em memória contém os 3 itens.
  2. **Cenário 2: Erro de Lógica (Estoque Insuficiente)**
     - Adicionar um item com 10 unidades (ex: "teclado").
     - Tentar remover 15 unidades desse item.
     - Verificar se o programa exibiu uma mensagem de erro clara.
     - Verificar se o *Inventario.dat* (e o estado em memória) ainda mostra 10 unidades.
     - Verificar se o *Auditoria.log* contém uma *LogEntry* com *StatusLog* (Falha ...).
  3. **Cenário 3: Geração de Relatório de Erros**
     - Após executar o Cenário 2, executar o comando *report*.
     - Verificar se o relatório gerado (especificamente pela função *logsDeErro*) exibe a entrada de *log* referente à falha registrada no Cenário 2 (a tentativa de remover estoque insuficiente).

5. Hospedagem no GitHub

O projeto deve ser hospedado em um repositório público no GitHub. O repositório deve ser organizado e conter:

- O código-fonte completo do sistema (.hs).
- Documentação (README.md) explicando como compilar e executar o programa no Online GDB ou no Repl.it, e um exemplo de uso dos comandos. A documentação deve conter o nome da instituição, disciplina, professor e nome dos alunos do grupo, em ordem alfabética, seguido do usuário de cada aluno no GitHub.
- No arquivo README.md deve existir um link para o ambiente de execução: GDB Online ou Repl.it
- O repositório deve ter um histórico de *commits* claro, e as contribuições de cada membro devem estar registradas. Atenção: a participação de cada aluno será validada por meio dos *logs* de interação no Github.

6. Divisão de Tarefas Sugerida

O trabalho pode ser dividido entre os quatro alunos, com cada um responsável por uma parte do sistema.

Aluno 1: Arquiteto de Dados

- **Responsabilidades:** Definir todos os tipos de dados (*Item*, *Inventario*, LogEntry, AcaoLog, StatusLog). Garantir que todos derivem Show e Read corretamente. Testar a serialização (read . show).
- **Entregável:** O módulo (ou seção do código) contendo as definições de tipos e instâncias de *Show*/*Read*.

Aluno 2: Lógica de Negócio Pura

- **Responsabilidades:** Desenvolver as funções puras de transação (*addItem*, *removeItem*, *updateQty*). Implementar a lógica de validação (*estoque*, *IDs* duplicados) e o retorno usando ***Either*** *String* *ResultadoOperacao*.
- **Entregável:** O conjunto de funções puras que formam o núcleo lógico do sistema, sem nenhuma operação de IO.

Aluno 3: Módulo de I/O e Persistência

- **Responsabilidades:** Implementar o *main :: IO ()* e o *loop* de estado. Implementar a lógica de **Inicialização** (carregar arquivos com *catch*) e **Sincronização** (chamar *writeFile* e *appendFile*). Implementar o *parser* de comandos do usuário.
- **Entregável:** As funções de IO, incluindo *main*, *loop*, e funções de leitura/escrita de arquivos.

Aluno 4: Validação, Documentação e Gerenciamento do Repositório

- **Responsabilidades:** Criar as funções de análise de logs (*historicoPorItem*, *logsDeErro*, etc.). Integrar o comando *report*. Manter o repositório no GitHub, escrever o *README.md*.
- **[Nova Responsabilidade]** Executar os cenários de teste manuais (seção 4.1) e documentar os resultados da execução no *README.md*.

- **Entregável:** As funções de análise, a integração do comando *report* e o *README.md* completo (incluindo a documentação dos testes).

7. Avaliação

O trabalho será avaliado com base nos seguintes critérios:

- **Lógica e Funcionalidade (70 pontos):**
  - Os tipos de dados são bem modelados e a serialização (*Show*/*Read*) funciona.
  - Os nomes de funções, arquivos e tipos de dados sugeridos neste documento foram utilizados no código.
  - Há uma separação clara entre funções puras (lógica) e impuras (IO).
  - O sistema é totalmente funcional: carrega, salva, audita e gera relatórios corretamente.
- **Organização e Legibilidade do Código (15 pontos):**
  - Código Haskell claro, comentado e bem estruturado (usando *where*, *let*, *case*, etc.).
  - Repositório GitHub organizado, com *README.md* claro e *commits* significativos.
- **Robustez (15 pontos):**
  - O sistema lida adequadamente com exceções de I/O (ex: *catch* na leitura de arquivos).
  - O sistema trata corretamente os erros de lógica de negócio (ex: estoque insuficiente).

7.1. Deméritos por Não Cumprimento

A nota final será impactada pelos seguintes deméritos, caso sejam identificados:

Não compilação: O código-fonte não compila no GDB Online (ou Repl.it). **Penalidade:** -100% da nota. O trabalho será zerado.
Indicação de Plágio: 30% ou mais de código idêntico a outro trabalho ou trabalhos com a mesma lógica porém identificadores diferentes. **Penalidade:** -100% da nota. O trabalho será zerado.
Mistura de Lógica Pura e Impura: Presença de operações de IO (ex: *writeFile*, *putStrLn*) dentro de funções puras da lógica de negócio (ex: *addItem*, *removeItem*). **Penalidade:** -40% da nota de "Lógica e Funcionalidade".
Falha na Persistência de Estado: O arquivo *Inventario.dat* não é salvo corretamente após operações de sucesso ou não é lido na inicialização. **Penalidade:** -30% da nota de "Lógica e Funcionalidade".
Falha no Log de Auditoria: O arquivo *Auditoria.log* não é atualizado (via *appendFile*) a cada operação. **Penalidade:** -30% da nota de "Lógica e Funcionalidade".
Não tratamento de exceção de I/O: O programa encerra com erro (*crash*) caso os arquivos de dados (*.dat* ou *.log*) não existam na primeira execução. **Penalidade:** -15% da nota de "Robustez".
Documentação de teste incompleta: O README.md não contém a documentação da execução dos cenários de teste manuais ou as outras informações obrigatórias. **Penalidade:** -30% da nota de "Robustez".
Dados de teste insuficientes: O sistema não foi populado com os 10 itens mínimos para teste.

Penalidade: -30% da nota de "Robustez".
Falha na Serialização/Desserialização: Os tipos não derivam *Show*/*Read* corretamente, impedindo o read dos arquivos de dados. **Penalidade:** -100% da nota de "Robustez".
Nomenclatura fora do padrão: Não utilizar os nomes de arquivos, tipos ou funções definidos na especificação. **Penalidade:** -50% da nota de "Organização e Legibilidade".

8. Entrega e Prova de Autoria

Um aluno do grupo será sorteado para responder a uma pergunta sobre o funcionamento ou implementação de qualquer parte do código. A falha na resposta implicará em uma redução de 35% na nota provisória do grupo.

***A nota provisória é soma da nota de Lógica e Funcionalidade (70 pontos) com a nota de Organização e Legibilidade do Código (15 pontos)* e a nota de *Robustez (15 pontos)* descontados os deméritos automaticamente aplicados.**
Se algum aluno do grupo faltar na prova de autoria, o grupo automaticamente perde os 35%.

O link para o repositório público no GitHub deve ser postado na data estipulada e apenas no local indicado na tarefa postada no canvas. ***Links enviados fora de data, ou por qualquer outro meio provocarão o zeramento do trabalho.***