UNIVERSIDADE FEDERAL DE UBERLÂNDIA

ANA LUÍZA MOREIRA SILVA

JEAN LUC GIRVENT DEU

MATHEUS MARTINS DE RESENDE

VINÍCIUS AUGUSTO DE SOUZA

PROGRAMAÇÃO FUNCIONAL

“Sistema de Gerenciamento de Biblioteca”

UBERLÂNDIA

2025

1. INTRODUÇÃO

Este trabalho teve como objetivo desenvolver um sistema de gerenciamento de biblioteca em **Haskell**, aplicando os conceitos de **programação funcional** estudados ao longo do semestre. O programa permite cadastrar livros e usuários, registrar empréstimos e devoluções, além de gerar relatórios sobre o acervo e as movimentações.

1.1 Contexto e Objetivos

O projeto foi desenvolvido para gerenciar uma biblioteca via terminal, com funções como cadastro de livros/usuários, controle de empréstimos e geração de relatórios gerais. Todos os dados são persistidos em arquivos para garantir continuidade entre diferentes execuções.

A implementação do código seguiu os princípios da programação funcional pura ensinados em salada de aula, evitando efeitos colaterais e utilizando recursão, funções de alta ordem e tipos algébricos como forma de organizar as nossas estruturas de dados.

1.2 Modularização do Código

O projeto foi dividido em módulos para facilitar a manutenção e o trabalho em equipe:

- Tipos.hs: Define as estruturas de dados (Usuário, Livro, Emprestimo)
- Funcoes.hs: Contém as funções puras para manipulação das listas (adição, remoção, busca)
- FuncoesInterface.hs: Gerencia a interação com o usuário (menus, entrada e saída de dados)
- Persistencia.hs: Cuida do carregamento e salvamento dos dados em arquivos
- Main.hs: Controla o fluxo principal do programa

1.3 Desafios e Soluções

Algumas das principais dificuldades encontradas foram:

- **Coordenação do trabalho em equipe:** Como cada membro desenvolveu principalmente um módulo diferente, foi necessário garantir que as funções se integrassem corretamente.
- **Tratamento de erros:** Validar entradas do usuário (como matrículas e códigos de livros) exigiu cuidado para evitar falhas no programa.
- **Persistência de dados:** Garantir que as informações fossem salvas corretamente em arquivos e recuperadas ao reiniciar o sistema.

A experiência adquirida no Trabalho 1 foi fundamental para superar esses desafios, permitindo um desenvolvimento mais ágil e organizado. Utilizando o fluxo geral do nosso projeto anterior como base, pudemos reaproveitar parte de suas funções puras – fazendo leves ajustes, o que facilitou a conclusão dessa atividade.

2. DIVISÃO DE TAREFAS

O desenvolvimento do sistema foi conduzido de forma colaborativa, com todos os membros contribuindo ativamente em diferentes etapas. A divisão de responsabilidades priorizou os interesses individuais e a integração entre os módulos. A tabela abaixo resume a distribuição principal:

Membro	**Responsabilidades**
Ana	Interface do usuário (FuncoesInterface.hs), mensagens e menus.
Jean	Persistência de dados (Persistencia.hs), README e instruções de uso.
Matheus	Testes, correção de erros, integração de módulos.
Vinícius	Funções puras (Funcoes.hs), relatório.

3. ORGANIZAÇÃO DO CÓDIGO

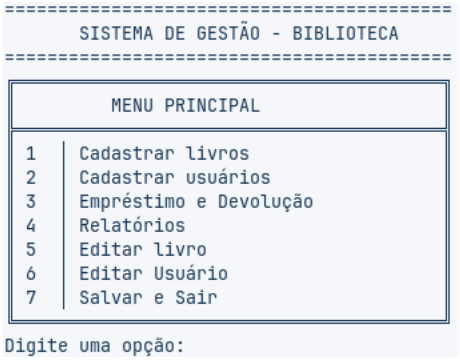
3.1 Módulo “Tipos.hs”

Define as estruturas de dados fundamentais usando tipos algébricos:

- Usuário: Campos como matrícula, nome, e-mail e histórico de empréstimos.
- Livro: Atributos como código, título, autor, status de disponibilidade e lista de espera.
- Emprestimos: Registra ID do empréstimo, código do livro e matrícula do usuário.

Características:

- Uso de *record syntax* para acesso claro aos campos.
- Derivação de Show, Read e Eq para serialização e comparação.



3.2. Módulo “Funções.hs”

Categoria	**Funções-Chave**	**Funcionalidade**
Tratamento de Erros	lerInteiro	Função segura para leitura de números inteiros.
CRUD	adicionarLivro, adicionarUsuário, adicionarEmpréstimo, removerUsuário, removerLivro, removerEmpréstimo	Inserção/remoção de elementos em listas.
Edição	editarLivroAno, editarLivroAutor, editarLivroTitulo, editarLivroStatus, editarUsuárioNome, editarUsuárioEmail	Atualização de atributos via recursão.
Gerenciamento	históricoUsuário	Atualiza o histórico de empréstimos de um usuário
Consultas	listaLivrosDisponíveis, listaLivrosIndisponíveis livroDisponível, listaLivroComReserva	Filtragem por status ou critérios específicos.
Lista de Espera	lEspera, buscaEspera, nomesListaEsp	Gerenciamento de reservas e verificações.
Identificação	acharMaiorCódigo, acharMaiorMatrícula, acharMaiorID, buscaAux	Geração de IDs únicos
Utilitários	buscaEmp, nomeLivro, nomeUsuário, nomeReserva	Conversão de dado e funções de auxílio

3.3. Módulo “FunçõesInterface.hs”

Funcionalidade	**Descrição**
Cadastro	adicionarLivroIO: Coleta dados, gera ID automático e persiste em arquivo.

	adicionarUsuárioIO: Registra usuários com matrícula única e salva dados.
Empréstimos	adicionarEmprestimoIO: Valida disponibilidade, registra empréstimo ou adiciona à lista de espera. removerEmprestimoIO: Gerencia devoluções, atualizando status do livro e removendo registros.
Relatórios	imprimirLivroIO: Exibe livros disponíveis/indisponíveis com formatação clara. listarEmprestimoIO: Lista todos os empréstimos ativos. históricoEmprestimoIO: Mostra histórico de empréstimos por usuário. esperaLivroIO: Exibe livros com lista de espera e usuários associados.
Menus	menuEmprestimosIO: Submenu com opções de empréstimo, devolução e consultas. menuRelatóriosIO: Menu para acesso a relatórios detalhados. editarUsuárioIO/editarLivroIO: Menus de edição de dados (nome, e-mail, autor, título, ano).
Edição	editarNomeIO/editarEmailIO: Atualiza nome e e-mail de usuários. editarAutorIO/editarTituloIO/editarAnoIO: Modifica atributos de livros.
Persistência	salvaBibliotecaIO: Salva estado completo do sistema (usuários, livros, empréstimos) em arquivo.

Técnicas utilizadas: Validação de entrada com readMaybe para evitar erros de tipo. Persistência automática em usuarios.txt, livros.txt, emprestimos.txt. Feedback visual (ex: ✔️ Livro adicionado com sucesso! ✖️ Código não encontrado. Verifique o código do livro).

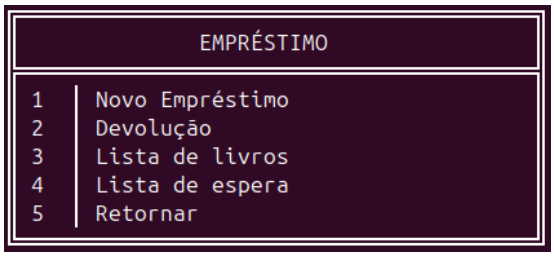
Tratamento de Erros: Verificação de listas vazias (ex: "Nenhum livro disponível"). Validação prévia de registros em operações críticas. Mensagens específicas (ex: "Código não encontrado"). Atualização simultânea de dados (empréstimos, status de livros, histórico).

Fluxo de Operações: Integração entre módulos (Funcoes.hs para lógica, Persistencia.hs para salvamento). Atualização em tempo real após ações (ex: adição/remoção de empréstimos). Menus interativos para navegação hierárquica.

3.4. Módulo “Persistencia.hs”

Responsável pela gestão de dados em arquivos:

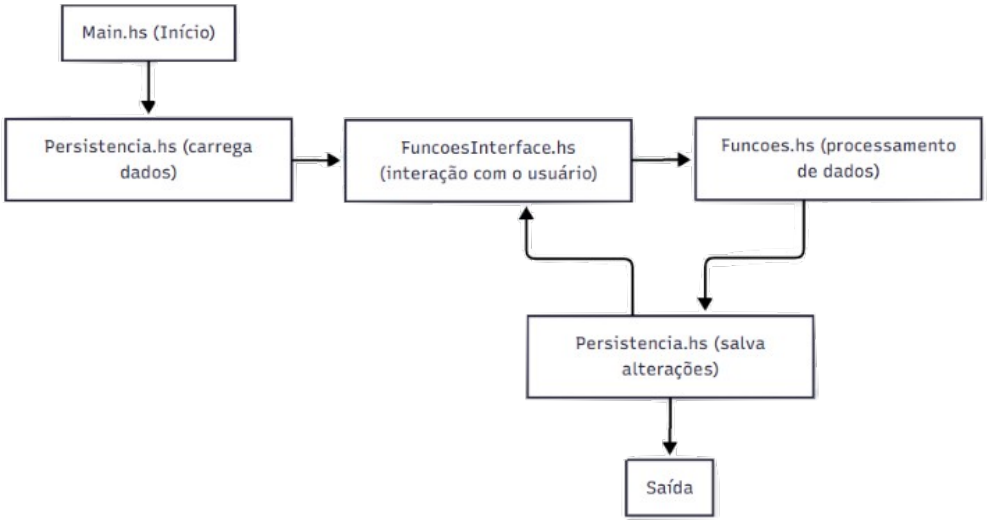
- **Serialização:** Conversão de estruturas para texto via show/read.
- **Tratamento de Erros:** Recuperação de arquivos vazios ou corrompidos.
- **Operações:** carregarUsuárioDeArquivo, salvarLivroEmArquivo, etc.



3.5. Módulo “Main.hs”

Coordena o fluxo do sistema:

1. **Inicialização:** Carrega dados dos arquivos.
2. **Loop Principal:**
    - Exibe menu com opções (cadastro, empréstimos, relatórios).
    - Chama funções específicas conforme a escolha do usuário.
3. **Integração:** Conecta todos os módulos, garantindo sincronia entre lógica, interface e persistência.



4. CONCLUSÃO

Como grupo que fez o "**Trabalho 1"** da disciplina juntos, usamos o que aprendemos para melhorar este novo projeto. A estrutura modular facilitou o processo de pegarmos partes do que já havíamos criado, especialmente algumas **funções puras** (como operações de CRUD e gerenciamento de listas), e adaptá-las as novas necessidades, o que tornou a implementação desse trabalho mais fluída e rápida. Essa reutilização garantiu uma maior consistência e confiabilidade no código, já que as funções já haviam sido testadas no trabalho anterior.

Ainda assim, trabalhar em grupo exigiu conciliar as **responsabilidades pessoais** (trabalho e outros projetos) com a alta demanda acadêmica do final do semestre (provas, listas de exercícios, apresentações e demais trabalhos de outras disciplinas). Nossos maiores desafios foram:

- **Integração de conhecimentos teóricos na prática:** Tivemos que aplicar conceitos de programação funcional que, embora estudados, eram pouco familiares na prática.
- **Sincronização de prazos:** Cada membro tinha ritmos e disponibilidades diferentes devido a outras obrigações.

Para resolver isso, nosso grupo focou em soluções simples:

1. **Comunicação constante**: Usamos o Discord para alinhar progressos e dúvidas conforme surgiam.
2. **Divisão clara de tarefas**: Cada um focou em módulos específicos (interface, lógica, persistência).
3. **Versionamento manual**: Compartilhávamos trechos do código em um grupo comum, atualizando conforme avançávamos.
4. **Testes contínuos**: Validávamos cada função isoladamente antes de integrá-las ao sistema.

Aprendemos que organizar o código em módulos separados facilita o trabalho em equipe, mas exige atenção redobrada na hora de integrar tudo. A experiência mostrou como a programação funcional ajuda a evitar erros inesperados, a criação de funções puras (que podem ser testadas isoladamente) facilita o processo de encontrar onde estão os erros e corrigi-los sem afetar diretamente outras partes do código. No final, o projeto funcionou bem e nos preparou para trabalhos colaborativos futuros.