CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
Faculdade de Tecnologia Baixada Santista Rubens Lara
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO
UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

SANTOS, SP

2025

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia Rubens Lara, como exigência para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Alexandre Garcia de Oliveira

SANTOS, SP

2025

AGRADECIMENTOS

Dedico este trabalho de conclusão de curso para a minha amada bruxa Beatrice.

Também o dedico a todos que me acompanharam nesta jornada e acreditaram em meu potencial. Em especial, agradeço a todo o pessoal da Semicollon Estúdios por terem sido o alicerce do desenvolvimento deste projeto, principalmente Victor Filipini e João Pedro Galvão Bargo durante o desenvolvimento de nossa *wiki*. Agradeço também a todos os irmãos da A∴L e da O∴O∴O por sempre me guiarem pelo caminho correto.

Agradeço a Fernanda, Marcos, Felipe Cannarozzo e todos os presentes nos "Rolês Romefeller". Estes momentos de alívio em períodos de estresse sempre me alegravam, e sinto que fiz bons companheiros que possuem um interesse em comum.

Agradeço também meu mentor, Alexandre Garcia, por me apresentar a linguagem incrível que é Haskell. Agradeço ao Walter Daniel, Dr. Alberto e Fernanda Erika por me fazerem dominar a magia da programação quando eu ainda não me sentia preparado. Sem vocês, eu não estaria aqui hoje.

Agradeço a minha namorada, Karina, por me ajudar com os diagramas de banco de dados, além de me apoiar neste período de trabalho e estudo com muito amor e carinho, e também com alguns puxões na orelha. Agradeço a minha deusa por me permitir entrar nesta faculdade com absoluta certeza. Agradeço a "Layla" por um dia ter dito que eu seria alguém grande, eu nunca esquecerei de você, jamais.

Por fim, agradeço a minha falecida irmãzona, a minha mãe, e meu pai.

Obrigado por todo o apoio.

RESUMO

Este trabalho apresenta o desenvolvimento de uma linguagem de marcação original denominada Markers. A linguagem foi projetada com foco em legibilidade, praticidade e acessibilidade para usuários que não possuem familiaridade com linguagens técnicas ou que buscam uma solução leve para organização de textos, estruturação de ideias e documentação. O processo envolveu desde a idealização da sintaxe até a implementação de um parser funcional que interpreta os elementos da linguagem e os converte para HTML. Como parte da aplicação prática, foi desenvolvido um ambiente de edição capaz de renderizar os arquivos .mks em tempo real, com funcionalidades como suporte a listas, links, imagens, blocos de código e referências. O trabalho também aborda os princípios de design da linguagem, os desafios técnicos na implementação e as possíveis aplicações futuras em ambientes educacionais, corporativos e de código aberto.

Palavras-chave: Linguagem de marcação, ABNT, Parser, Haskell, Markers

ABSTRACT

This work presents the development of an original markup language called Markers. The language was conceived with a focus on readability, practicality, and accessibility for users who are not familiar with technical languages or who seek a lightweight solution for organizing text, structuring ideas, and documentation. The development process included designing the syntax and implementing a functional parser that interprets the language's elements and converts them into HTML. As part of its practical application, an editing environment was developed to render .mks files in real time, with features such as support for lists, links, images, code blocks, and references. The work also discusses the language’s design principles, technical challenges in implementation, and potential future applications in educational, corporate, and open-source contexts.

Keywords: Markup language, ABNT, Parser, Haskell, Markers

LISTA DE ABREVIATURA E SIGLAS

LML	Linguagem de Marcação Leve
SGML	Standard Generalized Markup Language
HTML	Hypertext Markup Language
SAAS	Software as a Service (Software como um Serviço)
TCC	Trabalho de Conclusão de Curso
CLI	Command Line Interface (Interface de Linha de Comando)
CSS	Cascading Style Sheets
NBR	Norma Brasileira
ABNT	Associação Brasileira de Normas Técnicas
CSV	Comma Separated Values (Valores Separados por Vírgula)
GHC	Glasgow Haskell Compiler
ADT	Algebraic Data Type (Tipo de Dado Algébrico)
AST	Abstract Syntax Tree (Árvore Sintática Abstrata)
API	Application Programming Interface (Interface de Programação de Aplicações)
DOM	Document Object Model (Modelo de Objeto de Documento)
VPS	Virtual Private Server (Servidor Virtual Privado)
CRDT	Conflict-free Replicated Data Type (Tipo de Dado Replicado Sem Conflito)

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de linguagens de marcação tem se mostrado fundamental para a organização e a automatização de documentos em diversos contextos, especialmente no meio acadêmico e técnico. Linguagens como Markdown e LaTeX possibilitam que o conteúdo textual seja estruturado com clareza, facilitando a formatação. Porém, um dos problemas destas linguagens está em sua estrutura complexa e não-convidativa, não possuindo ampla gama de universalidade para variados outros tipos e estilos de documentos. Essas características mencionadas se mostram essenciais em trabalhos que exigem padronização, como artigos científicos, documentações técnicas e trabalhos de conclusão de curso (TCCs).

Apesar da ampla adoção de editores de texto proprietários como Google Docs e Microsoft 365, essas ferramentas apresentam limitações importantes. Entre elas, destacam-se a necessidade de conexão constante com a internet (no caso do Google Docs), o custo de licenciamento (no caso do Microsoft 365); além disso, essas plataformas não oferecem controle total sobre o processo de formatação e não se integram facilmente a fluxos de trabalho automatizados baseados em texto puro. Pode ser declarado injusto para um estudante ou pesquisador depender de ferramentas que exigem assinatura ou conexão constante, especialmente em contextos onde o acesso à internet é limitado ou inexistente.

Neste contexto, este trabalho apresenta a linguagem de marcação Markers, além de todo o seu ecossistema; concebida como uma alternativa livre, local e multiplataforma. O objetivo principal do projeto é fornecer uma ferramenta acessível, extensível e de fácil aprendizado, voltada para estudantes, profissionais e usuários independentes que necessitam de um ambiente eficiente para produção textual estruturada. A linguagem foi projetada para permitir a criação de documentos completos — incluindo títulos, subtítulos, listas, referências, imagens, códigos e fórmulas — com foco em compatibilidade com normas acadêmicas.

Por se tratar de um projeto baseado em software livre, o Markers permite que os usuários modifiquem e adaptem a ferramenta conforme suas necessidades, além de possibilitar seu uso sem custos. Essa abordagem visa atender especialmente o público que não dispõe de recursos para utilizar soluções comerciais ou que busca maior controle sobre o processo de produção textual.

O presente trabalho descreve o processo de concepção, implementação e validação da linguagem Markers, abordando suas motivações, arquitetura, recursos principais e casos de uso. Ao final, espera-se demonstrar que a solução proposta é viável, eficiente e relevante para diferentes perfis de usuários.

1.1 LINGUAGENS DE MARCAÇÃO

Uma linguagem de marcação é um conjunto de símbolos e regras que definem a estrutura e o formato de um documento. Estaremos trabalhando estritamente com linguagens de marcação leves (LML). As LMLs são tipos de linguagens de marcação que utilizam uma sintaxe simples e legível, tornando-se mais humanamente compreensível. Elas são projetadas para serem fáceis de usar e entender em forma de texto puro.

As linguagens de marcação tiveram suas origens na década de 1960, com o desenvolvimento do SGML (Standard Generalized Markup Language), que foi criado para descrever a estrutura de documentos eletrônicos. O SGML foi amplamente utilizado em aplicações científicas e técnicas, mas sua complexidade levou ao desenvolvimento de linguagens de marcação mais simples e específicas.

Um dos maiores exemplos é o HTML (Hypertext Markup Language), que é utilizado para criar páginas da web. O HTML é uma linguagem de marcação simples desenvolvida por Tim Berners-Lee - também responsável pela criação da World Wide Web em 1989 - em 1991, que permite a formatação de texto, imagens e links em um documento. O HTML surgiu originalmente para a publicação de documentos científicos, após receber especificações formais, ele se tornou um padrão aberto e amplamente utilizado na web. Este é uma linguagem de marcação baseada em tags, onde cada tag define um elemento do documento, como parágrafos, títulos, listas e links.

1.2 JUSTIFICATIVA

O desenvolvimento de uma sistema universal para gerar documentos acadêmicos é uma necessidade crescente na era digital. Com o aumento da produção e compartilhamento de conhecimento, a padronização e a acessibilidade dos documentos acadêmicos se tornaram questões cruciais. É necessário um software e uma formalização que agilize o processo da formatação e criação dos mesmos, respeitando todas as normas e diretrizes estabelecidas pelas instituições de ensino e publicações científicas de forma unânime. Este sistema será útil principalmente para auxiliar estudantes em trabalhos de conclusão de curso (TCC), além de ser uma ferramenta útil para professores e pesquisadores de todo o globo.

O desenvolvimento deste sistema também iria poupar o uso de aplicações de edição de texto rico de terceiros, como o Microsoft Word (Agora, Microsoft 365) e Google Docs, que agora estão no processo de migração, ou já migraram, para a metodologia de Software como um Serviço (SaaS). Devido ao fato deste sistema ser livre e de código aberto, ele poderá ser utilizado por qualquer pessoa, em qualquer lugar do mundo, sem a necessidade de pagar por

uma licença ou assinatura. Isso irá democratizar o acesso à tecnologia e permitir que mais pessoas possam criar e compartilhar conhecimento.

2 DESENVOLVIMENTO

Neste capítulo, descreveremos o processo de desenvolvimento da linguagem de marcação Markers, bem como a construção de seu parser, biblioteca funcional e interfaces de uso, incluindo uma aplicação web e uma ferramenta de linha de comando (CLI). Serão utilizadas ao todo 4 linguagens, sendo essas: Haskell, HTML, CSS e JavaScript.

A linguagem de marcação aqui desenvolvida tem como seu intuito permitir que os usuários possam formatar seus documentos de forma rápida e eficiente. A linguagem é baseada em tags, onde cada tag define um elemento do documento, como parágrafos, títulos, listas e links. As tags são delimitadas por caracteres entre parênteses e podem ter atributos adicionais para definir propriedades específicas do elemento.

O Parser é responsável por analisar o código-fonte escrito na linguagem de marcação e convertê-lo em um formato compreensível para o interpretador. Este Parser será escrito na linguagem Haskell, uma linguagem de programação puramente funcional e fortemente tipada, que é amplamente utilizada para o desenvolvimento de compiladores e interpretadores. O Haskell é uma linguagem de programação de alto nível, que permite a criação de programas complexos de forma concisa e legível. Utilizaremos das gloriosas implementações da linguagem e da comunidade para desenvolver nossa Arvore Sintática Abstrata (AST), que será responsável por armazenar a estrutura do documento e suas propriedades. Estaremos então transformando este Parser em uma biblioteca do Haskell, denominada de "The Markers Library", que poderá ser utilizada em outros projetos e aplicações. A biblioteca será projetada para ser fácil de usar e integrar com outras bibliotecas e ferramentas do Haskell, permitindo que os desenvolvedores possam criar aplicações personalizadas com facilidade.

Também sera desenvolvida uma aplicação de linha de comando (CLI) multiplataforma, que irá implementar a biblioteca "The Markers Library" e permitir que os usuários possam converter documentos escritos na linguagem de marcação para outros formatos, como HTML, Markdown, formatação ABNT e vários outros.

Também implementaremos a biblioteca "The Markers Library" em uma aplicação web, que permitirá que os usuários possam criar e editar documentos na linguagem de marcação diretamente em um navegador. A aplicação web será desenvolvida inteiramente em Haskell, utilizando o framework Yesod, uma das principais ferramentas para o desenvolvimento de aplicações web em Haskell. A aplicação web será projetada para ser fácil

de usar e intuitiva, mostrando a estrutura do documento em tempo real, permitindo que os usuários possam ver como o documento ficará formatado enquanto escrevem.

Por fim, será desenvolvida a aplicação "Miragem". Esta aplicação permitirá que os usuários possam salvar, editar e visuaizar seus documentos, além de compartilhar com outras pessoas, possibilitando a edição do mesmo em conjunto, em tempo real, facilitando a colaboração e o trabalho em equipe.

2.1 A LINGUAGEM MARKERS

A concepção da linguagem de marcação Markers surgiu a partir da identificação de uma lacuna no uso de linguagens de marcação existentes para documentos acadêmicos. A maioria das ferramentas disponíveis, como LaTeX, ABNTeX e Limarka, embora poderosas, apresenta barreiras de entrada significativas para usuários inexperientes exigindo conhecimento prévio em programação ou em sintaxes técnicas complexas, além de poucas vezes retornarem um resultado desejável, principalmente relacionado as normas presentes da ABNT.

A linguagem Markers foi idealizada como uma solução de sintaxe simplificada, com o objetivo de tornar a escrita e a formatação de documentos mais acessíveis e intuitivas. Sua estrutura baseada em tags legíveis visa proporcionar uma experiência mais fluida para o autor, permitindo a criação de textos com hierarquia, citações, sumário automático e outros elementos acadêmicos, sem a necessidade de comandos verbosos.

Neste capítulo exploraremos as características, normas e definições da linguagem, além de apresentar exemplos práticos de seu uso.

2.1.1 ESTRUTURA E DEFINIÇÃO DA LINGUAGEM

A estrutura da linguagem Markers é composta por uma série de tags que definem a formatação e a estrutura do documento. As tags são delimitadas por parênteses e podem conter atributos adicionais para definir propriedades específicas do elemento. As tags são iniciadas com um parêntese de abertura e terminadas com um parêntese de fechamento, e fechadas da mesma forma, porém com uma barra `/` antes do nome da tag. A sintaxe é semelhante à de outras linguagens de marcação, como HTML. Veremos um exemplo da abertura e fechamento de uma tag a seguir.

Um arquivo Markers é delimitado pela extensão de arquivo *".mks"*. Obrigatóriamente, o arquivo deve conter uma tag de título - sendo esta `(title)`, que define o título do documento. A tag `(title)` deve ser o primeiro elemento do arquivo, seguido pelo texto

(normal, ou formatado) que o usuário deseja inserir. A estrutura mais básica de um arquivo Markers é a seguinte:

```
(title)TÍTULO DO DOCUMENTO(/title)

Este é o conteúdo do documento.
```

2.1.1.1 TAGS DE FORMATAÇÃO DE TEXTO

A linguagem Markers possui uma série de tags que permitem a formatação de texto, como negrito, itálico, sublinhado e tachado. Essas tags são semelhantes às tags HTML, mas com uma sintaxe mais simples e legível. As tags de básicas de formatação de texto são as seguintes:

```
(c): Texto tachado.
(u): Texto sublinhado.
(b): Texto em negrito.
(i): Texto em itálico.
(k): Texto monoespaçado.
```

Por *design*, a linguagem Markers não irá suportar o aninhamento das tags de formatação de texto, ou seja, não será possível utilizar mais de uma tag de formatação em um mesmo trecho de texto. Devido sua natureza acadêmica, pode se considerar que o uso de mais de uma formatação em um mesmo trecho de texto não é uma boa prática, visto que pode gerar confusão e dificultar a leitura do documento.

As tags de formatação de texto são utilizadas da seguinte forma, observe o arquivo *.mks* de exemplo abaixo:

```
(title)TÍTULO DO DOCUMENTO(/title)

Este é um exemplo de texto (b)negrito(/b), (i)itálico(/i),
(u)sublinhado(/u) e (c)tachado(/c).
```

As tags mencionadas acima são as mais comuns, porém existem outras tags de formatação de texto, como a tag `(color)`, utilizada para definir a cor do texto, a tag `(n)` que define um texto negrito e itálico, além de outras.

Devido a natureza constantemente evolutiva da linguagem, novas tags podem ser adicionadas a qualquer momento, visando sempre melhorar a experiência do usuário e facilitar a formatação de documentos acadêmicos.

2.1.1.2 TAGS DE CORPO

As tags de corpo - também denominadas de tags principais - são as tags mais importantes da linguagem Markers. Elas são responsáveis por definir variadas estruturas e elementos do documento, como parágrafos, listas, tabelas, imagens e links.

Algumas de suas implementações foram diretamente inspiradas com as normas da ABNT (Associação Brasileira de Normas Técnicas) em mente, enquanto outras foram desenvolvidas para atender a necessidades específicas de formatação. Entraremos em detalhes sobre cada uma das tags principais.

2.1.1.2.1 TAGS DE USO COMUM

As tags de uso comum são as tags principais mais simples e básicas da linguagem Markers. Elas contam com diferentes variedades de formatação e estrutura. Por natureza, nenhuma dessas tags podem ser aninhadas, ou seja, não é possível utilizar mais de uma tag de uso comum em um mesmo trecho de texto. As tags de uso comum são as seguintes:

A tag `(link)` é utilizada para criar links dentro do documento. Ela é semelhante à tag *a* do HTML, mas com uma sintaxe mais simples e legível. A tag `(link)` deve conter dois atributos: o primeiro atributo é o texto que será exibido como link, e o segundo atributo é a URL para a qual o link irá apontar. é utilizada da seguinte forma:

```
(title)EXEMPLO DE LINK(/title)
Acesse o (link | https://www.google.com)Google(/link).
```

As tags `(img)` e `(localimg)` são utilizadas para inserir imagens no documento. A tag `(img)` é utilizada para inserir imagens de um URL externo, enquanto a tag `(localimg)` é utilizada para inserir imagens de um arquivo local. Ambas as tags devem conter um atributo que define o caminho da imagem. A tag `(img)` é utilizada da seguinte forma:

```
(title)EXEMPLO DE IMAGEM(/title)
(img | cat.jpg)lolcat.(/img)
```

Já a tag `(localimg)` é utilizada da seguinte forma:

```
(title)EXEMPLO DE IMAGEM LOCAL(/title)
(localimg | images/cat.jpg)lolcat.(/localimg)
```

A tag `(code)` é usada para definir blocos de código no documento. O seu conteúdo utilizará de fonte monoespaçada e será sensível à quebras de linha. Observe o exemplo:

```
(title)EXEMPLO DE CÓDIGO(/title)

(code)
    public static void main(String[] args) {
        System.out.printLn("Olá, mundo!");
    }
(/code)
```

O `(ref)` é usado para referências em parte de um texto ou em um parágrafo completo. Ela é separada em cinco seções seguindo a ordem determinada na norma NBR 6023 da ABNT, sendo essas, em ordem: link do conteúdo, autor, título, ano e data de acesso. Confira o exemplo:

```
(title)EXEMPLO DE REFERÊNCIA(/title)

(ref | https://example.com/ | Autor | Conteúdo | 2025 | 01/01/2025)
Esta é uma referência. O mesmo é um exemplo.
(/ref)
```

A tag `(quote)` é utilizada para citações. Ela necessita da informação do autor da citação como atributo, sendo seguida do conteúdo. Observe:

```
(title)EXEMPLO DE CITAÇÃO(/title)

(quote | AUTOR)
"Uma frase que o autor disse."
(/quote)
```

A tag `(table)` é utilizada para criar tabelas dentro do documento. Ela segue um formato diferente das tags de tabelas que encontramos em outras linguagens de marcação, sua sintaxe é inspirada na formatação de arquivos CSV (Comma Separated Values). A tag `(table)` é definida pela abertura e fechamento da tag, seu conteúdo é definido pela abertura e fechamento de parenteses, com o conteúdo separado pelo caractere *pipe* `|` . A primeira linha sempre representa o cabeçalho da tabela, seguido das colunas. Observe o exemplo:

```
(title)EXEMPLO DE TABELA(/title)

(table)
    ( Nome   | Idade | Cidade  )
    ( Miguel |  20   | Santos  )
    ( João   |  21   | Taubaté )
    ( Victor |  20   | Lorena  )
(/table)
```

O `(hr)` é uma tag de marcação de um separador, seja uma separação horizontal (horizontal rule) similar ao `hr` do HTML, quanto uma quebra de página. Essa tag não possúi atributos.

```
(title)EXEMPLO DE SEPARAÇÃO(/title)

Este é o conteúdo de uma página.
(hr)
Este é o conteúdo de outra página.
```

A tag `(summary)` é uma tag de marcação de um sumário, que será gerado automaticamente com base nos capítulos presentes no documento. Não é necessário definir manualmente cada um dos capítulos dentro dela. Esta tag recebe apenas o nome desejado do sumário como atributo.

```
(title)EXEMPLO DE SUMÁRIO(/title)

(summary | SUMÁRIO)
```

2.1.1.2.2 TAGS RECURSIVAS

As tags recursivas são outro tipo presente das tags de corpo. Estas mesmas marcações possuem a propriedade exclusiva de poderem ser aninhadas dentro de sí mesmas. Este exemplo ficará mais claro quando vermos os exemplos de seu uso. As duas marcações recursivas presentes na linguagem são `(chap)` - usada para determinar capítulos - e (>>), usada para determinar listas.

Em documentos acadêmicos, é comum aninharmos capítulos dentro de outros capítulos, criando seções específicas para cada elemento do trabalho; sendo assim, por natureza, as tags referentes à determinação destes capítulos deve ser recursiva.

A tag `(chap)` determina a criação de um novo capítulo no documento. Existem duas maneiras de determinar um capítulo: não paginado ou paginado. Para criar um novo capĩtulo

não paginado, é necessário apenas de um único atributo, sendo esse o nome do capítulo.

```
(title)EXEMPLO DE CAPÍTULO(/title)

Este é um conteúdo.
(chap | CAPITULO 2)
    Este é o conteúdo do capítulo 2.
(/chap)
```

Para criar um capítulo paginado, utilizamos dois atributos, a página e o nome do capítulo. Utilizamos os capítulos paginados em documentos acadêmicos que necessitam da página declarada no sumário, ou em algum outro equivalente.

```
(title)EXEMPLO DE CAPÍTULO(/title)

Este é um conteúdo.
(chap | 1 | CAPITULO 2)
    Este é o conteúdo do capítulo 2.
(/chap)
```

Dado sua natureza recursiva, podemos aninhar capítulos dentro de capítulos, criando assim subcapítulos. Observe:

```
(title)EXEMPLO: CAPÍTULOS ANINHADOS(/title)

Este é um conteúdo fora do capítulo.
(chap | CAPÍTULO 1)
    Este é o conteúdo do capítulo 1.
    (chap | CAPÍTULO 1.1)
    Este é o conteúdo do subcapítulo 1.1.
    (/chap)
(/chap)
```

A tag `(>>)`, chamada de *arrow* (seta) ou *list* (lista) determina uma lista que pode ser expandida ou fechada (em inglês, *toggle list*). É utilizada para organizar informações de forma não intrusiva para o leitor, fazendo seu conteúdo ser de opcional leitura. Definimos-a com um atributo, sendo este o nome da lista.

```
(title)EXEMPLO DE LISTA(/title)

Este é um conteúdo do corpo do documento.
(>> | LISTA 1)
    Este é o conteúdo dentro da lista.
    (>> | LISTA 1.1)
        Esta é outra lista, recursiva.
    (/>>)
(/>>)
```

2.1.1.3 TAGS DE USO ESPECIAL

As tags de uso especial são tags separadas das definições de corpo do documento de um arquivo *.mks*, sendo assim principalmente utilizadas para definição de metadados para a exportação do documento.

A tag `(meta)` é usada para definir dados do próprio documento em sí, tal como o autor, subtítulo, local de escrita, ano de escrita e outros dados importantes para identificação. Esta tag possui a propriedade de ter outras tags dentro de sí que determinam cada um destes valores. É uma prática comum sempre definir a tag `(meta)` depois do título do documento.

```
(title)EXEMPLO DE TAG META(/title)
(meta)
    (institution)FACULDADE DE TECNOLOGIA RUBENS LARA(/institution)
    (author)MIGUEL AGUIAR(/author)
    (subtitle)EXEMPLO E USO DA TAG(/subtitle)
    (location)SANTOS - SP(/location)
    (year)2025(/year)
(/meta)
```

Por mais que seja uma tag de corpo, a tag `(math)` é considerada uma tag especial por utilizar uma linguagem própria dentro de sí mesma. Esta linguagem é denominada *Euler*. É determinada pela abertura da tag, o código em Euler, e o fechamento da mesma. Iremos ver as definições da linguagem Euler e sua implementação no capítulo 2.2.5.

```
(math)
 [prod | k=1 | m | (a_k)]
(/math)
```

Outra tag especial é o comentário. Os comentários estão presentes apenas no arquivo fonte do `.mks` e não influenciam a estrutura do documento. Um comentário pode ser útil para anotações internas do próprio autor sobre o documento. É determinado pela abertura de parenteses, seguido por dois traços e o conteúdo.

```
(title)EXEMPLO DE COMENTÁRIO(/title)
(-- Nota: Isso é um comentário, remover depois. --)
Este é um texto que será mudado no futuro.
```

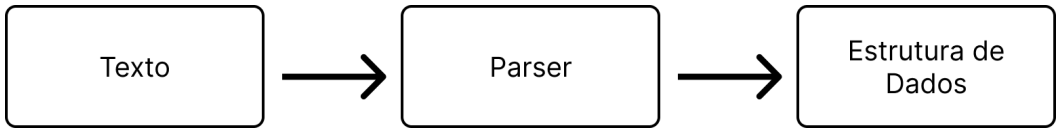
2.2 DESENVOLVIMENTO DO PARSER E INTERPRETADOR

Tendo a estrutura e regras da linguagem Markers definida, podemos começar o desenvolvimento do analisador sintático (parser) e interpretador. Utilizaremos primariamente

a linguagem puramente funcional Haskell junto da biblioteca *megaparsec*, *bytestring*, *base64-bytestring* e *text* para seu desenvolvimento. Será necessário o uso das ferramentas GHC (Glasgow Haskell Compiler, o compilador da linguagem Haskell) e Cabal - gerenciador de pacotes do Haskell. Usaremos as linguagens HTML, CSS e JavaScript no processo de interpretação e tradução da linguagem para outros formatos. Também será utilizado as ferramentas Git - para versionamento, e Visual Studio Code como editor primário de texto.

O Parser é responsável por analisar um texto e determinar sua estrutura lógica. Sua função é transformar este texto em uma estrutura de dados (comumente, uma árvore), que irá capturar a hierarquia implícita no texto e será conveniente para processamento futuro.

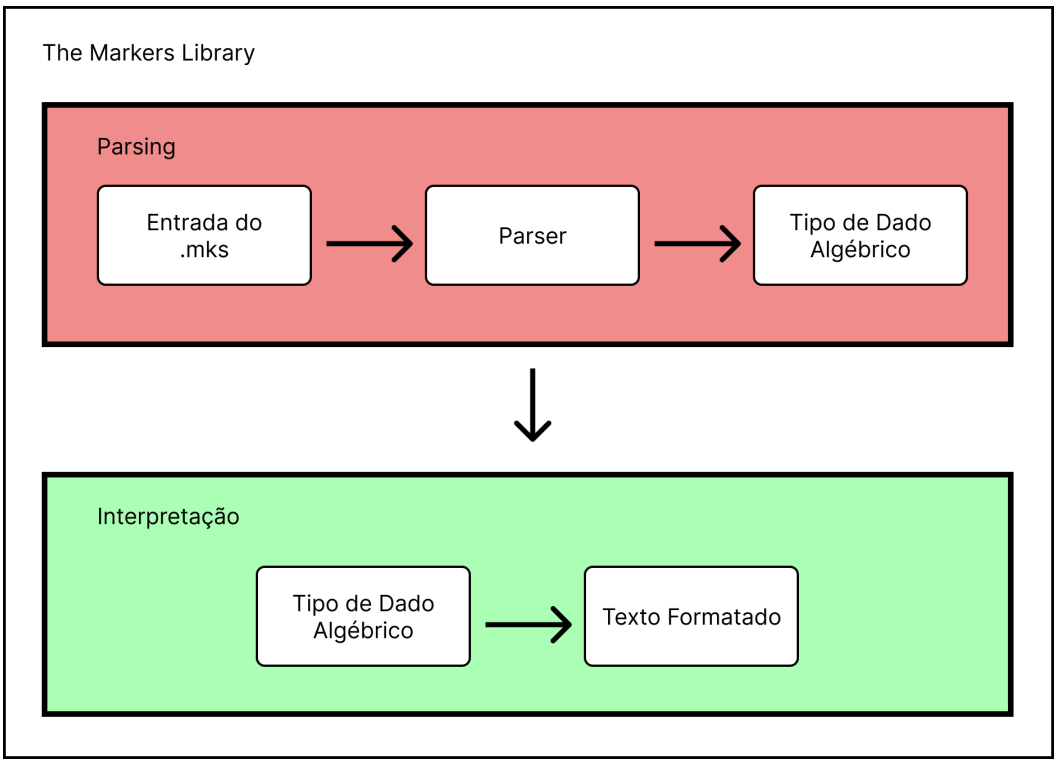
FIGURA 1 – Fluxo de um Parser.



Fonte: Autor, 2025

Neste caso, o texto que será analisado sintáticamente será um arquivo `.mks`, que será traduzido para os respectivos Tipos de Dados Algébricos (ADT) que iremos definir em nossa Árvore de Sintáxe Abstrata (AST). Entraremos em detalhes sobre estas etapas nos capítulos seguintes. Os ADTs respectivos serão então processados (ou, interpretados) e transformados em outros tipos de texto, mantendo o seu conteúdo original intacto. Iremos nomear o software que irá realizar este processo de *The Markers Library* (A Biblioteca do Markers). Foi optado este nome devido ao fato que o mesmo será disponibilizado livremente como uma biblioteca de interpretação da linguagem.

FIGURA 2 – Fluxo do The Markers Library.



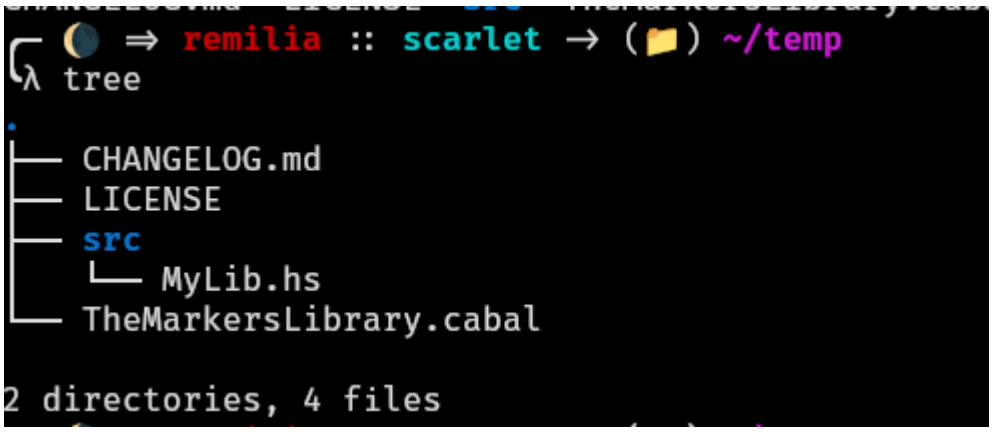
Fonte: Autor, 2025

2.2.1 PREPARAÇÕES PARA O DESENVOLVIMENTO

Iremos começar fazendo o *download* do Glasgow Haskell Compiler via a ferramenta *GHCup* (disponível em https://www.haskell.org/ghcup/) e do Visual Studio Code (disponível em https://code.visualstudio.com/). O GHCup irá instalar o compilador, e também será o responsável por instalar o gerenciador de pacotes *Cabal* em nossa máquina. Para mantermos o versionamento do projeto, instalaremos o Git (disponível em https://git-scm.com/downloads).

Com todos os preparativos feitos, será criado uma nova pasta dedicada ao desenvolvimento do projeto. Esta pasta irá se chamar "TheMarkersLibrary". Iremos inicializar um projeto do Haskell nesta pasta utilizando o Cabal com o comando `cabal init` em um terminal. Este comando irá gerar para nós a estrutura base de um projeto de biblioteca em Haskell. Sendo a seguinte sua estrutura:

FIGURA 3 – Estrutura de arquivos do Cabal.



Fonte: Autor, 2025

Ao abrirmos o arquivo `TheMarkersLibrary.cabal`, devemos alterar a linha `build-depends` e adicionar as bibliotecas que serão necessárias para o projeto, transformando:

```
build-depends: base >= 4.14 && < 5
```

No seguinte trecho:

```
build-depends: base >= 4.14 && < 5,
               megaparsec,
               bytestring,
               base64-bytestring,
               text
```

Podemos então apagar o arquivo `src/MyLib.hs`. É de se notar que a pasta `src` é onde manteremos todo o código fonte de nossa aplicação, onde os separaremos em varios `modules`. Com tudo isso feito corretamente, podemos seguir para o desenvolvimento da Árvore Sintática Abstrata.

2.2.2 ARVORE SINTÁTICA ABSTRATA

A Árvore Sintática Abstrata (AST) é uma representação hierárquica da estrutura de um programa ou documento. Ela captura a relação entre os diferentes elementos do código, permitindo que o analisador sintático compreenda a lógica e a semântica do texto. A AST é composta por nós, onde cada nó representa um elemento da linguagem, como expressões, declarações e estruturas de controle.

A AST é uma forma simplificada que captura a lógica do código, permitindo que ferramentas de desenvolvimento realizem operações como análise estática, otimização e

transformação de código.

Definiremos a AST da linguagem Markers utilizando Tipos de Dados Algébricos (ADT) do Haskell. Os ADTs são uma forma de definir tipos de dados complexos em Haskell, permitindo a criação de estruturas de dados personalizadas. Eles são amplamente utilizados na programação funcional para representar dados estruturados e hierárquicos. Nossa AST será composta por diferentes tipos de nós, cada um representando um elemento da linguagem Markers. Notaremos que grande parte da AST é composta por tipos de dados recursivos, ou seja, um tipo de dado que se refere a si mesmo. Isso é útil para representar estruturas de dados que podem ter um número variável de elementos, como listas e árvores.

A estrutura de nossa AST será criada no arquivo `src/AbstractSyntaxTree.hs` ; no arquivo, adicionaremos a linha `module Ast.AbstractSyntaxTree where` seguida das respectivas estruturas de dados que definiremos no capítulo seguinte.

2.2.2.1 TIPOS DE DADOS ALGÉBRICOS

Os Tipos de Dados Algébricos (ADT) são uma forma de definir tipos de dados complexos em Haskell, permitindo a criação de estruturas de dados personalizadas. Eles são amplamente utilizados na programação funcional para representar dados estruturados e hierárquicos. A seguir, definiremos os principais tipos de dados que compõem a AST da linguagem Markers.

Um exemplo de um tipo de dado algébrico seria um tipo `Texto` . Ele é um tipo de dado que pode ser `Formatado` ou `NãoFormatado` . O tipo `Texto` é definido no Haskell da seguinte forma:

```
data Texto = Formatado String | NãoFormatado String
    deriving (Show, Eq)
```

Neste exemplo, o tipo `Texto` é definido como um tipo algébrico que pode ser `Formatado` ou `NãoFormatado` . Ambos os tipos `Formatado` e `NãoFormatado` são estruturas que "encapsulam" uma *String* (cadeia de caracteres), que é o texto em si.

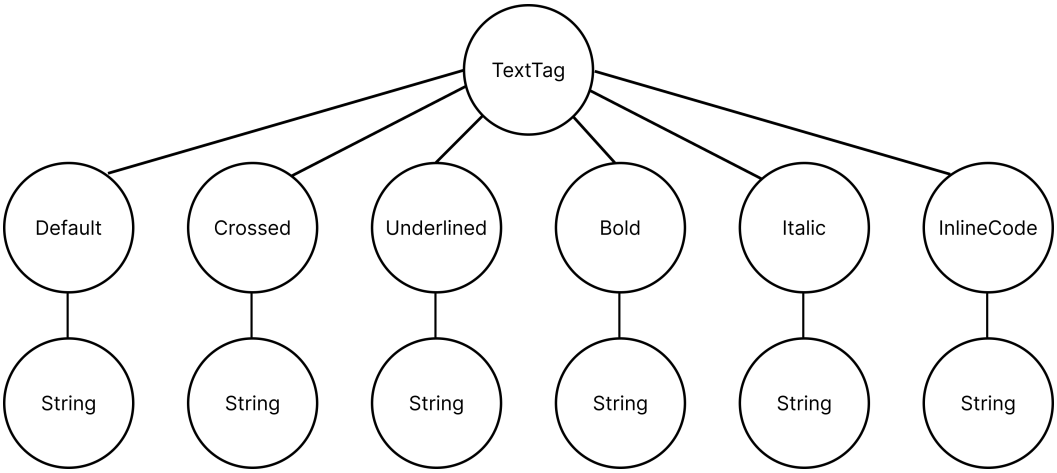
O tipo `Texto` é um exemplo de um tipo de dado algébrico, pois ele pode ter múltiplas formas (ou construtores), cada uma representando uma variação do tipo.

2.2.2.2 DEFINIÇÃO DA ESTRUTURA DA ÁRVORE

Começaremos definindo os tipos de dados que irão representar as tags de formatação de texto da linguagem Markers. Esta abstração utilizará um tipo de dado algébrico, onde cada

construtor representa uma tag de formatação. Este ADT irá se chamar `TextTag`, seus construtores representarão os vários tipos possíveis de tags de formatação que existem na linguagem.

FIGURA 4 – Árvore representativa do ADT TextTag.



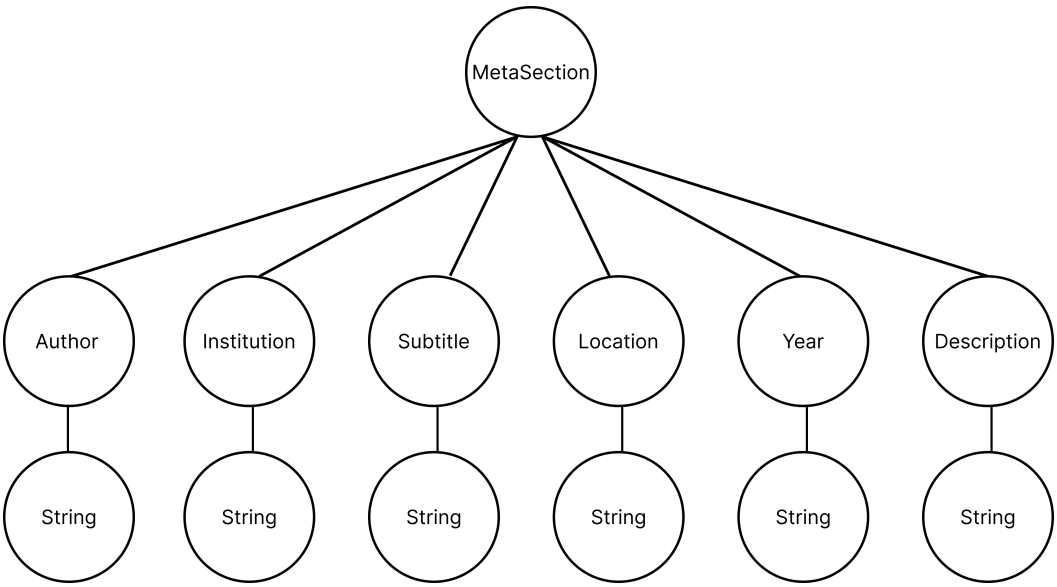
Fonte: Autor, 2025

A representação deste grafo em Haskell será feita da mesma forma que definimos o tipo `Texto` no capítulo 2.2.2.1, onde cada construtor representa uma tag de formatação. Sendo assim, declaramos o tipo de dado `TextTag` seguido de cada construtor representativo de uma tag de formatação. Sendo assim;

```
data TextTag = Default String
             | Crossed String
             | Underlined String
             | Bold String
             | Italic String
             | CodeInline String
             -- Outras formatações...
            deriving (Show)
```

A mesma lógica será aplicada para definirmos o tipo de dado `MetaSection`. O tipo de dado referente à tag `(meta)`. Sendo assim representada como:

FIGURA 5 – Árvore representativa do ADT MetaSection.



Fonte: Autor, 2025

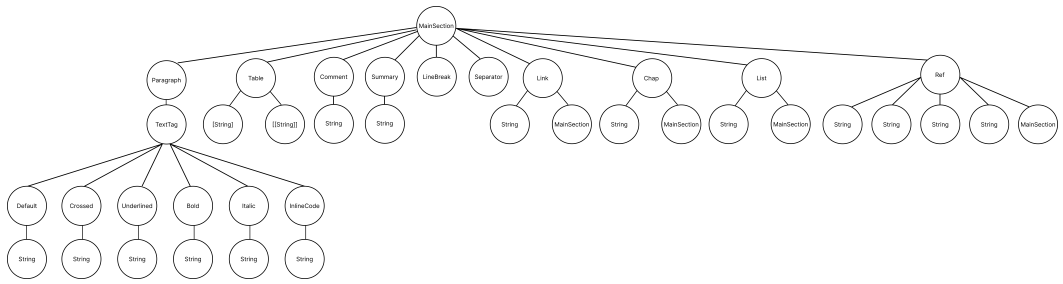
Novamente, sua representação em Haskell será feita seguinte a mesma lógica que definimos o tipo `TextTag`, onde cada construtor representa uma tag de formatação. Sendo assim, declaramos o tipo de dado `MetaSection` seguido de cada construtor representativo de uma tag de formatação. Sendo assim;

```
data MetaSection = Author String
                 | Institution String
                 | Subtitle String
                 | Location String
                 | Year String
                 | Description String
                 deriving (Show)
```

Para representarmos as tags de corpo da linguagem Markers, criaremos o tipo de dado `MainSection`. Esta será a nossa estrutura principal de nossa AST. Note que grande parte de seus nós são recursivos. A explicação disso se dá na lógica das tags utilizadas. Utilizando a tag `(chap)` como um exemplo, devemos ter acesso a todas as outras tags disponíveis no corpo do documento dentro dela.

Isso se dá pelo fato de que um capítulo pode conter outros capítulos, e dentro destes capítulos podem existir outras tags, logo, por sua definição, esta é uma tag que necessita da recursividade da `MainSection`. Observe o exemplo do grafo abaixo:

FIGURA 6 – Árvore representativa do ADT MainSection.

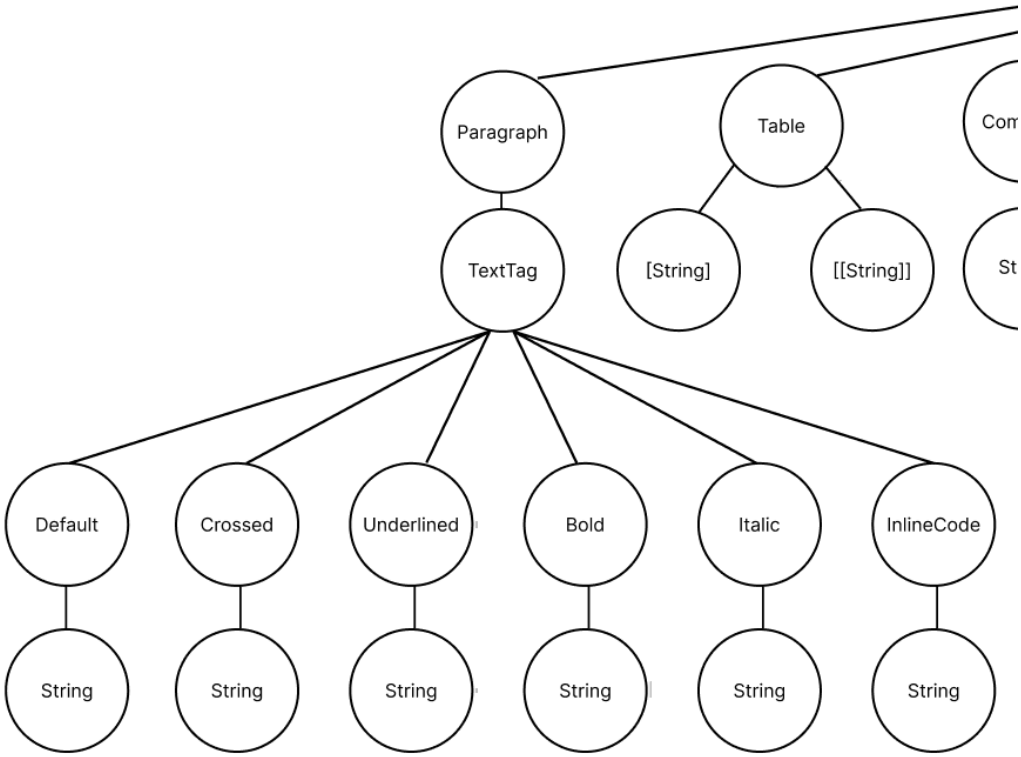


Fonte: Autor, 2025

Mesmo que a nossa AST permita a inclusão de qualquer tipo de dado algébrico dentro de uma tag que possua a propriedade recursiva de `MainTag`, para aquelas que estritamente não possuem essa regra, tal como a tag `(link)`, iremos limitar isso no momento de seu parsing. Faremos isso adiante no capítulo 2.2.3. Note que o grafo acima não representa a natureza completa da estrutura do ADT que faremos no Haskell, já que sua representação não é linear, sendo efetivamente infinita.

Também podemos notar que o tipo de dado `TextTag` está encapsulado em um construtor `Paragraph`, que representa um parágrafo, podemos fazer isso para melhor organização do código, abstraindo a lógica de formatação de alguma tag específica que possua vários tipos de formatação ou de uso, mas que ainda mantém uma lógica de formatação específica.

FIGURA 7 – Secção "Paragraph" da ADT MainSection.



Fonte: Autor, 2025

Representaremos esta estrutura em nosso arquivo da seguinte maneira:

```
data MainSection =
            Empty
            | Paragraph TextTag
            | Meta [MetaSection]
            | List String [MainSection]
            | Chap String [MainSection]
            | Ref String String String String String [MainSection]
            | Link String [MainSection]
            | Image String String [MainSection]
            | ImageUrl String [MainSection]
            | Video String [MainSection]
            | Audio String [MainSection]
            | Table [String] [[String]]
            | Code [MainSection]
            | Quote String [MainSection]
            | Summary String
            | Abntchapter String String [MainSection]
            | Commentary String
            | LineBreak
            | Tab
            | Separator
            deriving (Show)
```

Note que definimos a recursividade das tags necessárias (sendo essas, aquelas que podem possuir tanto outros conteúdos formatados quanto sí mesmas) como um `[MainSection]`, isto é, uma lista de elementos do tipo `MainSection`. Isso nos permite criar uma estrutura de árvore, onde cada nó pode conter outros nós, formando assim uma hierarquia de elementos. Caso não declaramos a recursividade em uma lista, seria possível adicionar apenas um único elemento de `MainSection` em seu conteúdo, não permitindo com que o aninhamento de - por exemplo - capítulos dentro de capítulos que possuem parágrafos com formatação de texto fosse possível.

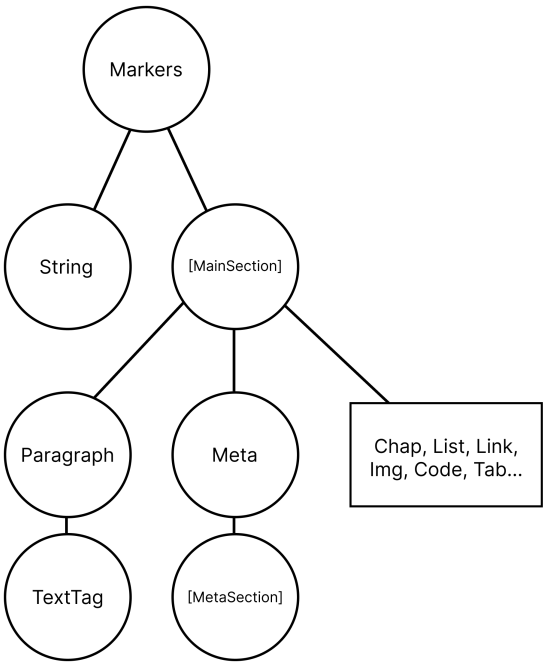
Isto irá se tornar mais claro quando formos implementar o parser no capítulo 2.2.3, onde veremos exemplos detalhados de como os tipos de dados algébricos são tratados quando possuem atributos e conteúdos.

Por fim, criaremos mais um tipo de dado algébrico nomeado `Markers`. Este será o responsável pela estrutura base do documento, consistindo de apenas duas seções representando `(title)` e o seu conteúdo. Sendo assim, ela possuirá um único construtor denominado `MarkersMain`, recebendo uma *string* para seu título e um `[MainSection]` para seu conteúdo. Sua definição será:

```
data Markers = MarkersMain String [MainSection]
    deriving (Show)
```

Tendo todas as estruturas definidas, podemos demonstrar que a representação mais simples de nossa arvore sintática abstrata pode ser dada com o seguinte grafo, contendo um nó raiz e dois nós filhos, sendo um uma folha (sem filhos) do tipo `String` e outro um nó com `n` nós filhos do tipo `[MainSection]`:

FIGURA 8 – Estrutura da Árvore Sintática Abstrata.



Fonte: Autor, 2025

2.2.3 ANÁLISE SINTÁTICA E CONVERSÃO

Realizaremos o desenvolvimento do parser utilizando a biblioteca *megaparsec*. Esta biblioteca é uma ferramenta de combinador de analisadores monádicos desenvolvida para efetuar análise sintática em Haskell, permitindo a criação de parsers eficientes e flexíveis. A biblioteca *megaparsec* é amplamente utilizada na comunidade Haskell e oferece uma ampla gama de recursos para facilitar o desenvolvimento de parsers.

O nosso parser será separado em dois `modules`, sendo eles `Paragraphs.hs` - que irá cuidar do *parsing* de tags de texto - e `MainSection.hs` - que irá cuidar do *parsing* de tags de corpo.

Começaremos definindo o parser de tags de texto no arquivo `src/Paragraphs.hs`. Este parser será responsável por analisar as tags de formatação de texto e convertê-las em nós da árvore sintática abstrata.

```
module Parser.Paragraphs where

import Text.Megaparsec
import Text.Megaparsec.Char
import Data.Void
```

```
import Control.Applicative

type Parser = Parsec Void String
```

Com a definição do *megaparsec* feita, podemos iniciar cada parser individual de cada tag do Markers. Iniciaremos criando uma função chamada `parseBold`, a mesma será definida da seguinte forma:

```
parseBold :: Parser MainSection
parseBold = do
    _ <- string "(b)"
    content <- manyTill anySingle (string "(/b)")
    return (Paragraph (Bold content))
```

A função `parseBold` é uma mônada que representa o parser da tag `(b)`, ele retorna um nó do tipo `MainSection`, sendo este um nó do tipo `Paragraph Bold`. Esta função realiza as seguintes operações em ordem:

```
1. Lê a tag de abertura (b) e ignora seu conteúdo.
2. Lê o conteúdo da tag (b) (manyTill) até encontrar a
   tag de fechamento (/b) (anySingle (string "(/b)").
3. Retorna um nó do tipo MainSection
   com o conteúdo lido.
4. O conteúdo lido é encapsulado em um nó do
   tipo Bold, que representa a formatação em negrito.
```

Ou seja, quando temos o texto `(b)Texto em Negrito(/b)`, o parser irá realizar a conversão desta parte do texto para o tipo de dado algébrico `Paragraph (Bold "Texto em Negrito")`. A mesma lógica do parser pode ser aplicada para as tags `(i)` (itálico), `(u)` (sublinhado), `(c)` (tachado) e etc, mudando as tags de abertura e fechamento, e o tipo de dado algébrico que encapsula o conteúdo lido para seu membro respectivo na Árvore Sintática Abstrata.

Para tags que não possuem conteúdo, como a tag `Separator` (`(hr)`), criaremos um parser que apenas lê a única tag de abertura e ignora seu conteúdo, retornando diretamente o tipo de dado algébrico respectivo a tag. Sendo assim, o parser de `(hr)` é definido como:

```
parseSeparator :: Parser MainSection
parseSeparator = do
    _ <- string "(hr)"
    return Separator
```

Para tags que necessitam de um atributo, como a tag `(color)`, devemos separar a tag e o atributo por um *pipe* `|`. Isso pode ser feito da seguinte maneira:

```
parseColor :: Parser MainSection
parseColor = do
    _ <- string "(color |"
    color <- manyTill anySingle (string ")")
    content <- manyTill anySingle (string "(/color)")
    return (Paragraph (Color color content))
```

A função `parseColor` é uma mônada que representa o parser da tag `(color)`, ela retorna um nó do tipo `MainSection`, sendo este um nó do tipo `Paragraph Color`. Primeiramente é lida a tag de abertura até a separação com o caractere `|`, após isso, lê-se o atributo (a cor) até o fechamento da tag de abertura. Depois, é lido o conteúdo interno da tag até o fechamento da tag de formatação. Por fim, retorna-se um nó do tipo `MainSection` com o conteúdo lido, encapsulado em um nó do tipo `Color`, que representa a formatação de cor.

Ou seja, se tivermos o texto "`(color | #FBC099)Texto alaranjado(/color)`", o parser irá realizar a conversão desta parte do texto para o tipo de dado algébrico `Paragraph (Color "#FBC099" "Texto alaranjado")`.

Dado a definição de documento da linguagem Markers descrito no capítulo 2.1.1, o corpo do documento cujo não possúa nenhuma tag de formatação será automaticamente lido como um texto comum, ou seja, um nó do tipo `Paragraph Default` com o conteúdo lido. No momento que qualquer outra tag é lida, o tipo de dado `Paragraph Default` deve ser parado e sucedido pelo nó respectivo da tag lida. Isso é feito através da função `parseDefault`.

Esta função lê todo o conteúdo do documento até encontrar uma tag de formatação ou uma quebra de linha. Este é dado pelo uso da chamada de função `someTill anySingle (void (lookAhead tagStart) <|> eof)`. A chamada da função `lookAhead tagStart` verifica se o próximo caractere é uma tag de formatação, enquanto a chamada da função `eof` verifica se o final do arquivo foi atingido. Caso nenhuma das duas condições seja satisfeita, a função `someTill` irá continuar lendo o conteúdo do documento até que uma delas seja satisfeita, retornando por fim um `Paragraph (Default content)`.

Note que a função `void` é utilizada para descartar o resultado da chamada da função `lookAhead`, já que não precisamos do valor retornado, apenas de saber se a condição foi satisfeita. A função `tagStart` é definida na própria função `parseDefault`, e é responsável por verificar se o próximo caractere é uma tag de formatação definida. Sempre que uma nova tag de formatação for adicionada, devemos adicionar uma nova chamada de função `try` dentro da função `tagStart`. A função `try` é utilizada para tentar fazer o parsing de uma tag de formatação, e caso não seja possível, ela irá falhar e continuar com o parsing normal do documento.

```
parseDefault :: Parser MainSection
parseDefault = do
content <- someTill anySingle (void (lookAhead tagStart) <|> eof)
return (Paragraph (Default content))
where
    tagStart :: Parser String
    tagStart = choice
    [ try (string "(c)")
    , try (string "(u)")
    , try (string "(b)")
    , try (string "(i)")
    , try (string "(k)")
    , -- ...
    ]
```

Note o uso do operador `<|>`, este é um operador de escolha do megaparsec - similar a um `OR`. Ele tenta fazer o parsing da primeira opção (neste caso, a chamada da função `lookAhead`), e caso não seja possível, ele tenta fazer o parsing da segunda opção (neste caso, o final do arquivo). Caso nenhuma das duas opções seja possível, a função `someTill` irá falhar e retornar um erro. Utilizaremos este operador com frequência em nosso parser, já que ele nos permite fazer escolhas entre diferentes opções de parsing.

Após definirmos cada parser individual de cada tag, podemos fazer uma função de parsing generalizada que irá chamar cada um dos parsers individuais. Esta função irá ser responsável por fazer o parsing do corpo do documento inteiro com todas as tags de formatação de texto.

```
parseContent :: Parser MainSection
parseContent = parseBold
               <|> try parseItalic
               <|> try parseUnderlined
               <|> try parseCrossed
               <|> try parseDefault
```

A função `parseContent` irá tentar executar cada um dos parsers individualmente sob uma string, em ordem de prioridade de execução. Caso o primeiro parser falhe, ele irá tentar o segundo, e assim por diante. Caso nenhum dos parsers seja possível, a função `parseContent` irá falhar e retornar um erro. Note que isso apenas generaliza qual parser será utilizado na string lida, e não corresponde a todo o conteúdo, encerrando assim que um parser for executado com sucesso. Para isso, devemos criar uma função que irá chamar a função `parseContent` até que o final do arquivo seja atingido. Esta função será chamada de `parseParagraph`.

```
parseParagraph :: Parser [MainSection]
```

```
parseParagraph = many parseContent
```

A função `parseParagraph` irá chamar a função `parseContent` até que o final do arquivo seja atingido, retornando uma lista de nós do tipo `MainSection` . Esta lista irá conter todos os nós do tipo `MainSection` lidos no documento. Como estamos por hora trabalhando apenas com tags de texto, todas estas tags retornadas serão do tipo `Paragraph` . Podemos ver o retorno da lista dos tipos de dado utilizando a função `parseTest` do megaparsec. Esta função irá executar o parser e imprimir o resultado na tela.

Rodando a função `parseTest parseParagraph "Isso é um (b)teste(/b) de (i)parsing(/i)."` , obtemos:

```
[Paragraph (Default "Isso \233 um "),Paragraph (Bold "teste"),
 Paragraph (Default " de "),Paragraph (Italic "parsing"),
 Paragraph (Default ".")]
```

Isso significa que o parser conseguiu ler o conteúdo do documento com sucesso, onde cada parte do documento foi corretamente formatado, observe:

```
Isso é um      : Paragraph (Default "Isso \233 um ")
(b)teste(/b)   : Paragraph (Bold "teste")
de             : Paragraph (Default " de ")
(i)parsing(/i) : Paragraph (Italic "parsing")
.              : Paragraph (Default ".")
```

Com isso, notamos o porque é necessário o retorno de nosso `parseParagraph` ser um `[MainSection]` , já que devemos armazenar cada uma das tags de formatação em ordem de leitura. Assim mantendo a formatação do texto original, já que a ordem das tags de formatação é importante para a interpretação correta do documento.

2.2.3.1 ANÁLISE SINTÁTICA DE CORPO

A análise sintática do corpo do documento é feita de forma semelhante à análise sintática das tags de formatação de texto. A diferença está no que fora mencionado no capítulo 2.2.2.2, onde iremos limitar o uso de tags dentro de tags que não possuem a propriedade recursiva. Utilizaremos das funções de parsing similares a `parseContent` diretamente dentro das funções de parsing das tags que devemos limitar. Por exemplo, a tag `(link)` não pode conter tags de formatação de texto dentro dela, então devemos limitar o uso de tags dentro da mesma. Isso é feito da seguinte maneira:

```
parseLink :: Parser MainSection
parseLink = do
```

```
    _ <- string "(link | "
    url <- manyTill anySingle (string ")")
    content <- parseStrictDefault "(/link)"
    _ <- string "(/link)"
    return (Link url content)
```

Note o uso da função `parseStrictDefault` ao coletar o conteúdo da tag, esta função é responsável por permitir que apenas texto comum, sem formatação ou qualquer outra tag, seja lido dentro da tag `(link)` (note que estamos importando `Parsers.Paragraphs` no módulo `MainTags.hs` para permitir isto). Observe sua definição:

```
parseStrictDefault :: String -> Parser [MainSection]
parseStrictDefault st = manyTill parseDefault
                        (lookAhead (string st))
```

A função `parseStrictDefault` irá chamar a função `parseDefault` até que o final do arquivo seja atingido, ou até que a string passada como parâmetro seja lida. Isso garante que apenas texto comum, sem formatação ou qualquer outra tag, seja lido. Esta função é uma função universal que pode ser aplicada em qualquer outra tag cujo devemos limitar a liberdade de formatação de texto de acordo com a definição da linguagem Markers dada no capítulo 2.1.

Um uso interessante do `parseStrictDefault` está na tag `(code)`. Como não podemos permitir o parsing incorreto dentro desta tag, já que devemos mostrar código-fonte válido dentro dela, podemos utilizar a função `parseStrictDefault` para garantir que apenas texto comum seja lido dentro dela. Ou seja, mesmo quando algo poderia ser uma tag válida, ela não será lida como tal. Isso garante que o código-fonte dentro da tag `(code)` seja lido corretamente, sem formatação ou qualquer outra tag.

Ou seja, a string `"Isso é um (b)teste(/b)"` será convertida para o ADT `Paragraph (Default "Isso \233 um (b)teste(/b)")`.

```
parseCode :: Parser MainSection
parseCode = do
    _ <- string "(code)"

    -- Permite uso de tags do Markers dentro de sí
    -- Pois é definida como Paragraph Default.
    content <- parseStrictDefault "(/code)"
    _ <- string "(/code)"
    return (Code content)
```

O oposto pode ser aplicado para as tags `(>>)` e `(chap)`, onde necessitamos de absolutamente toda a recursividade possível. Para isso, podemos definir estas funções com

uma chamada de caixa preta que realiza o parsing do conteúdo utilizando de `parseMainContent` para efetuar o parsing recursivo de todas as tags que podem estar presentes no mesmo. Observe:

```
parseList :: Parser MainSection
parseList = do
    _     <- string "(>> |"
    title <- manyTill anySingle (string ")")
    content <- parseListBody "(/>>)"
    _     <- string "(/>>)"
    return (List title content)

    where
    parseListBody :: String -> Parser [MainSection]
    parseListBody stopMark =
        manyTill parseMainContent (lookAhead (string stopMark))
```

Aqui, o conteúdo da lista será determinado por um `parseListBody` passando o parâmetro da tag de fechamento da lista. O `parseListBody` é uma função de caixa preta que recebe uma "marcação de parada" (similar ao nosso `parseStrictDefault`), porém efetua o parsing de todo o conteúdo do corpo do documento, permitindo assim que qualquer tag que possua a propriedade recursiva de `MainSection` seja lida dentro dela, ela efetua isso chamando `manyTill parseMainContent` até que a tag de texto `stopMark` (sendo este, o `(/>>)` passado anteriormente) seja encontrada.

No caso de nossa tag `(chap)`, esta deve ter uma modularidade a mais no momento de seu parsing, como definido no capítulo 2.1.1.2.2 A tag `(chap)` pode ter atributos a mais que mudam o retorno de seu ADT, podendo ser `Chap` ou `Abntchapter` no momento de sua conversão. Podemos realizar isso utilizando da função `optional` do megaparsec, que irá permitir que o parser continue mesmo que a tag não possua o atributo. A função `optional` é uma função do megaparsec que permite que um parser seja opcional, ou seja, ele pode ser lido ou não. Isso é útil para tags que podem ter atributos opcionais.

```
parseChap :: Parser MainSection
parseChap = do
    _     <- string "(chap |"
    mNum  <- optional $ try (do
             num <- manyTill anySingle (string " | ")
             return num)
    title <- manyTill anySingle (string ")")
    content <- parseChapBody "(/chap)"
    _     <- string "(/chap)"
    return $ case mNum of
             Just number -> Abntchapter number title content
             Nothing     -> Chap title content

where
```

```
parseChapBody :: String -> Parser [MainSection]
parseChapBody stopMark =
    manyTill parseMainContent (lookAhead (string stopMark))
```

Acima, o atributo `mNum` utiliza do optional para receber um possível atributo. No retorno da função, utilizamos da sintaxe `case-of` do Haskell para verificar se o atributo foi lido ou não. Caso tenha sido lido, retornamos o ADT `Abntchapter` com o número do capítulo, título e conteúdo. Caso contrário, retornamos o ADT `Chap` com o título e conteúdo. Note também que assim como em nossas listas com o `parseListBody`, o corpo do capítulo também é lido utilizando a função `parseMainContent` na função `parseChapBody`, permitindo assim que qualquer tag que possua a propriedade recursiva de `MainSection` seja lida dentro dela.

As propriedades de metadados possuem a propriedade única de só poderem ser lidas dentro de uma tag `(meta)`, para realizarmos isso, utilizamos uma lógica similar ao uso do `parseStrictDefault`, porém utilizando os parsers de cada uma das tags de metadados. Assim, o parser da tag `(meta)` é definido como:

```
parseMeta :: Parser MainSection
parseMeta = do
    _ <- string "(meta)"
    _ <- many (char ' ' <|> char '\n')
    content <- manyTill parseMetaContent (string "(/meta)")
    _ <- many (char ' ' <|> char '\n')
    return (Meta content)
```

Onde a função `parseMetaContent` é definida como:

```
parseMetaContent :: Parser MetaSection
parseMetaContent = parseAuthor
                   <|> parseInstitution
                   <|> parseSubtitle
                   <|> parseLocation
                   <|> parseYear
                   <|> parseDescription
```

Permitindo que apenas tags como `(author)`, `(institution)`, `(subtitle)`, `(location)`, `(year)` e `(description)` sejam lidas dentro dela, e nunca fora dela. Utilizar de um parser separado semânticamente desta forma permite que o código fique mais organizado e fácil de entender, além de permitir que cada parser tenha sua própria lógica de parsing, caso necessário.

Outra tag que possui uma propriedade única no momento de seu parsing é a tag `(localimg)`. Esta tag difere de `(img)`, já que ela não possui um atributo de URL, e sim um

atributo de caminho local. Esta tag deve converter no momento do parsing o caminho da imagem passada para Base 64, para que o arquivo possa ser lido diretamente do disco rígido, permitindo a universalidade no compartilhamento de um documento offline.

Para realizar este feito, primeiramente definiremos a função `convertToBase64`, esta função receberá um caminho de arquivo e irá retornar uma `IO String`.

```
convertToBase64 :: FilePath -> IO String
convertToBase64 path = do
    bytes <- BS.readFile path
    return $ unpack $ decodeUtf8 (B64.encode bytes)
```

A função `convertToBase64` utiliza a biblioteca *bytestring* para ler o arquivo, e a biblioteca *base64* para codificá-lo em Base 64. A função `unpack` é utilizada para converter o resultado de volta para uma string.

A função `convertToBase64` é uma função de entrada/saída (IO), pois ela realiza operações de leitura de arquivos, que são operações que podem falhar. Portanto, devemos utilizar a função `unsafePerformIO` para executar essa função dentro do parser. A função `unsafePerformIO` é uma função do Haskell que permite executar uma ação de entrada/saída dentro de um contexto puro. Isso deve ser utilizado com cautela, pois pode levar a comportamentos inesperados se não for usado corretamente.

```
parseLocalImage :: Parser MainSection
parseLocalImage = do
    _        <- string "(localimg | "
    resource <- manyTill anySingle (string ")")
    let extension = takeExtension resource
    content  <- parseStrictDefault "(/localimg)"
    _        <- string "(/localimg)"
    let b64res = unsafePerformIO (convertToBase64 resource)
    return (Image b64res extension content)
```

Assim, a função `parseLocalImage` recebe o caminho do arquivo, lê o conteúdo do arquivo utilizando a função `convertToBase64`, e retorna um nó do tipo `Image` com o conteúdo lido. Note que a função `takeExtension` é utilizada para pegar a extensão do arquivo, para que possamos retornar o tipo correto de imagem no momento de conversão. Isto fará mais sentido quando definirmos o interpretador da linguagem Markers no capítulo 3.2.1, onde iremos converter o arquivo para HTML.

A tag `(table)` possui uma sintaxe diferente de todas as outras tags de corpo definidas até então. Sua definição (como dada no capítulo 2.1.1.2.1) determina a existência de um cabeçalho e de várias linhas - como também demonstrada em sua definição na Árvore

Sintática Abstrata, sendo esta `Table [String] [[String]]`, onde temos uma linha que determina o cabeçalho, e uma lista de linhas que determinam o seu corpo. Cada linha é composta por `n` colunas. A primeira lista (`[String]`) representará o cabeçalho da tabela, enquanto a lista de listas (`[[String]]`) representarão as linhas da tabela. Cada linha será composta por várias colunas, separadas por um caractere `|`.

A primeira definição será o nosso parser, que irá ler a tag `(table)`, como uma tag normal. Após isso, será descartada uma quebra de linha e espaços em branco, permitindo o uso de várias linhas em branco entre o cabeçalho e as linhas da tabela.

```
parseTable :: Parser MainSection
parseTable = do
    _ <- string "(table)"
    _ <- many (char ' ' <|> char '\n')
```

Após isso, o cabeçalho será lido utilizando a função `parseTableContent`, que irá ler a primeira linha da tabela e separá-la em colunas. Esta será definida como outra função de caixa preta dentro da função `parseTable`. Esta função descarta a chave de abertura, e recebe o cabeçalho da tabela como uma string completa. Assim, descartamos a chave de fechamento.

```
parseTableContent :: Parser [String]
parseTableContent = do
    _ <- string "("
    header <- manyTill anySingle (string ")")
    _ <- many (char ' ' <|> char '\n')
```

Nosso retorno deve filtrar os caracteres `|` e usá-lo como delimitador para separar as colunas. Isso é feito utilizando a função `splitOn` da biblioteca *bytestring*. Isto será aplicado com o uso da função `pack` para transformar a string em um tipo de dado `ByteString`. A função `splitOn` irá separar a string em uma lista de `ByteString`.

Por fim, a função `unpack` é aplicada em cada *bytestring* utilizando a função `fmap` para transformar cada um em uma string, retornando uma lista de strings, por causa disso, o retorno de nossa função é um `[String]`.

```
parseTableContent :: Parser [String]
parseTableContent = do
    _ <- string "("
    header <- manyTill anySingle (string ")")
    _ <- many (char ' ' <|> char '\n')
    return $ fmap unpack (splitOn (pack " | ") (pack header))
```

O corpo da tabela também será lido utilizando a função `parseTableContent` , com a diferença que será usada juntamente à função `manyTill` do megaparsec. Isso irá permitir que várias linhas sejam lidas, retornando uma lista de listas de strings, até que a tag `(/table)` seja encontrada e descartada.

```
parseTable :: Parser MainSection
parseTable = do
    _ <- string "(table)"
    _ <- many (char ' ' <|> char '\n')
    header <- parseTableContent
    _ <- many (char ' ' <|> char '\n')
    rows <- manyTill parseTableContent (string "(/table)")
    return (Table header rows)

    where
    parseTableContent :: Parser [String]
    parseTableContent = do
        _ <- string "("
        header <- manyTill anySingle (string ")")
        _ <- many (char ' ' <|> char '\n')
        return $ fmap unpack (splitOn (pack " | ") (pack header))
```

Será então retornado o ADT `Table` com os conteúdos lidos. Podemos observar o retorno do parser utilizando a função `parseTest` novamente:

```
ghci> parseTest parseTable "(table)
                      (a | b | c)
                      (d | e | f)
                      (h | i | j)
                      (/table)"
```

Obtemos assim, o retorno `Table ["a","b","c"] [["d","e","f"], ["h","i","j"]]` , simbolizando corretamente a separação de nosso cabeçalho com as linhas da tabela.

Após implementarmos a função `parseMainContent` que irá chamar cada um dos parsers de corpo definidos, podemos finalizar o parser de corpo do documento. A função `parseMainContent` é definida da seguinte maneira:

```
parseMainContent :: Parser MainSection
parseMainContent =  parseCommentary
                    <|> parseTable
                    <|> parseQuote
                    <|> parseChap
                    <|> parseSummary
                    <|> parseRef
                    <|> parseList
                    <|> parseLink
```

```
        <|> parseTrace
        <|> parseImageUrl
        <|> parseImage
        <|> parseVideo
        <|> parseAudio
        <|> parseCode
        <|> parseMeta
        <|> parseContent
```

Com isto concluido, podemos finalizar todo o processo de parsing de um arquivo Markers. No arquivo `Markers.hs`, criaremos nosso último parser, que será dedicado ao título e conteúdo do arquivo como definido no capítulo 2.1.1. Seu código será o seguinte:

```
parseTitle :: Parser String
parseTitle = do
    _ <- string "(title)"
    manyTill anySingle (string "(/title)")

parseMarkers :: Parser Markers
parseMarkers = do
    title <- parseTitle
    content <- manyTill parseMainContent eof
    return (MarkersMain title content)
```

Podemos então testar nossa função `parseMarkers` utilizando a função `parseTest` do megaparsec. Utilizando a seguinte entrada:

```
(title)Isso é um arquivo Markers(/title)
Ele é um arquivo (b)customizável(/b) que será convertido.
```

Teremos o retorno do tipo de dado `MarkersMain`:

```
MarkersMain "Isso \233 um arquivo Markers"
        [LineBreak,
        Paragraph (Default "Ele \233 um arquivo "),
        Paragraph (Bold "customiz\225vel"),
        Paragraph (Default " que ser\225 convertido.")]
```

Com isto, temos a estrutura de nosso parser completa e funcional. Podemos observar que o parser conseguiu ler o título e o conteúdo do arquivo corretamente, retornando um nó do tipo `MarkersMain` com o título e o conteúdo lido. O conteúdo lido é uma lista de nós do tipo `MainSection`, onde cada nó representa uma parte do documento. Agora, uma vez que temos o parser completo, podemos utilizá-lo para ler qualquer arquivo Markers e convertê-lo para os tipos de dados algébricos da Árvore Sintática Abstrata.

2.2.4 INTERPRETADOR

Nosso interpretador será responsável por converter os tipos de dados algébricos de nossa árvore sintática abstrata que foram préviamente *parseados* para o formato de saída desejado. Neste caso, o formato de saída será o HTML. Iniciaremos desenvolvendo um interpretador simples que irá apenas retornar um HTML crú com o conteúdo lido. Declararemos em `Markers.hs` a função `parseFileWith` e a função `convertToRaw`.

A função `parseFileWith` será uma função de alta ordem que receberá uma função que converte `Markers` para `String`, sendo assim definida como uma função `(Markers → String)`, recebe outra String, e por fim, retorna uma String.

```
parseFileWith :: (Markers -> String) -> String -> String
parseFileWith renderer text = case parse parseMarkers "" text of
    Left err -> errorBundlePretty err
    Right res -> renderer res
```

Esta função recebe a função `renderer`, que no nosso caso, será a `convertToRaw`. Ela então aplicará o parser `parseMarkers` na string lida, e caso não ocorra nenhum erro, ela irá aplicar a função `renderer` no resultado do parser, retornando o resultado da conversão. Nesta função é utilizada a mônada Either do Haskell, que é uma forma de lidar com erros de forma segura. Caso ocorra um erro, a função irá retornar uma mensagem de erro formatada utilizando a função `errorBundlePretty` do megaparsec.

A função `convertToRaw` será responsável por converter o tipo de dado `Markers` para uma string HTML. Ela será definida da seguinte maneira:

```
convertToRaw :: String -> String
convertToRaw = parseFileWith toRaw
```

A função `convertToRaw` irá chamar a função `parseFileWith` passando a função `toRaw` como parâmetro. A função `toRaw` será responsável por converter o tipo de dado `Markers` para uma string HTML. A função `toRaw` agirá como o interpretador da linguagem Markers, convertendo os tipos de dados algébricos da Árvore Sintática Abstrata para a sintaxe HTML.

Esta função será definida no módulo `Converter/To.hs`. Começaremos definindo-a da seguinte maneira:

```
toRaw :: Markers -> String
toRaw (MarkersMain title sections) =
```

```
  "<h1>" <> title <> "</h1>"
  <> foldr (\x acc -> helper x <> acc) "" sections
```

Aqui, definimos que a função irá receber um tipo de dado `Markers`, e irá retornar uma string. Aplicamos o *pattern matching* para pegar o título do arquivo e o conteúdo lido. O título é encapsulado em uma tag `<h1>` e o conteúdo lido é passado para a função `helper`. Esta função será responsável por converter cada nó do tipo `MainSection` para uma string HTML. A função `foldr` é utilizada para iterar sobre a lista de nós do tipo `MainSection`, aplicando a função `helper` em cada um deles.

A função `helper` também seguirá a lógica de aplicar o *pattern matching* para cada nó do tipo `MainSection`. Ela será definida como uma função de caixa preta dentro de `toRaw`, tendo seu tipo de retono como `MainSection → String`. Por início, iremos definir o conteúdo de texto padrão, as tags de formatação de texto.

O texto padrão `(Paragraph (Default content))` deve ser retornado apenas como sí mesmo, sem qualquer tag HTML envolvendo-o. Podemos definir isso da seguinte forma:

```
helper :: MainSection -> String
helper (Paragraph (Default content)) = content
```

Podemos aplicar a mesma lógica que fizemos no título do arquivo para as tags que possúam formatação. Envonvendo o seu conteúdo nas tags HTML correspondentes. Assim, podemos definir as tags de formatação de texto da seguinte maneira:

```
helper :: MainSection -> String
helper (Paragraph (Default content))   = content
helper (Paragraph (Bold content))      = "<b>" <> content<> "</b>"
helper (Paragraph (Italic content))    = "<em>" <> content <> "</em>"
helper (Paragraph (Crossed content))   = "<s>" <> content <> "</s>"
-- Outras tags de formatação de texto...
```

Para as tags que necessitam de parâmetros adicionais, podemos apenas inserí-los no local onde o conteúdo é lido no HTML. Por exemplo, a tag `(color)` pode ser interpretada e convertida da seguinte maneira:

```
helper (Paragraph (Color color content)) = "<b><span
                                           style=\"color: "
                                           <> color <> "\" >"
                                           <> content
                                           <> "</span></b>"
```

44

No caso das tags de corpo, como `(list)`, `(table)` e `(chap)`, seguiremos aplicando a mesma lógica, porém seu conteúdo irá necessitar de uma chamada recursiva da função `helper` para tratar cada nó individual dentro de cada um deles. Por exemplo, a tag `(list)` pode ser convertida da seguinte maneira:

```
helper (List title content)
    = "\n\n"  title  "\n\n"
     ""  foldr (\x acc -> helper x  acc) "" content
     ""
     "\n"
```

Como o conteúdo de `List` é uma lista de nós do tipo `MainSection`, devemos aplicar a função `helper` em cada um utilizando o `foldr` explicado anteriormente. O mesmo se aplica para as outras tags de corpo, com poucas diferenças sendo levadas em consideração.

Podemos também observar a diferença intrinseca entre as tags `(img)` e `(localimg)` no momento de interpretação, observe:

```
helper (Image base64String mimeType content)
    = "\n helper x  acc) "" content
     "\">\n"

helper (ImageUrl url content)
    = "\n helper x  acc) "" content
     "\">\n"
```

Podemos também observar como nossa tabela é constuida, iterando por cada elemento `String` de seu cabeçalho e seu corpo.

```
helper (Table headers rows)
    = "\n"
     "\n"
     "\n"
    -- Iterando pelo cabeçalho.
     foldr (\x acc -> ""  x  ""  acc) "" headers
     "\n"
     "\n"
     "\n"
    -- Iterando por cada lista de String no corpo.
     foldr (\x acc -> "\n"
     foldr (\y xcc -> ""  y  ""  xcc) "" x
     "\n"  acc) "" rows
     "\n"
```

```
<> "</table>\n"
```

Continuaremos definindo a interpretação de cada tag até que todas as tags de formatação e corpo estejam definidas. Por fim, fecharemos nosso *pattern matching* com a verificação que ocorre quando nenhum caso é satisfeito. Isso será útil para quando o interpretador falhar, e não conseguir ler o arquivo corretamente. Neste caso, devemos retornar uma mensagem de erro informando que a tag não foi reconhecida. Isso pode ser feito da seguinte maneira:

```
helper (LineBreak)
    = "\n<br>\n"

helper _ = "<!-- Esta tag não foi reconhecida -->\n"
```

Note que por estarmos retornando um HTML, não temos impedimento de adicionarmos CSS ou JavaScript customizado. Isto será útil para quando definirmos o interpretador para o formato da norma ABNT no capítulo 2.2.4.3.

Após todos os casos terem sidos definidos, podemos finalizar o interpretador de HTML com a função `toRaw`. Podemos então prosseguir para outros interpretadores que seguem a mesma funcionalidade, porém com saídas diferentes. Por exemplo, o interpretador para ABNT, que irá converter o arquivo Markers para o formato da norma ABNT, ou o interpretador para Markdown, que irá converter o arquivo Markers para o formato Markdown.

As possibilidades para o interpretador são infinitas, e podemos criar interpretadores para qualquer formato que desejarmos. O importante é que o interpretador siga a mesma lógica de conversão, onde cada tag é convertida para o formato desejado, e o conteúdo é lido e convertido de acordo com as regras do formato. É semântica do Markers em sua definição de código-aberto que caso um interpretador não esteja presente, ou alguma tag não esteja disponível, que ela seja adicionada de imediato para manter a universalidade da linguagem.

Podemos testar nosso interpretador chamando a função `convertToRaw` com o conteúdo Markers que desejamos, e o resultado será o HTML gerado. Por exemplo, chamando a função `convertToRaw` com o seguinte conteúdo:

```
convertToRaw "(title)Isso é um arquivo Markers(/title)
              Ola, (b)mundo!(/b)"
```

Irá retornar diretamente a string HTML:

```
"<h1>Isso \233 um arquivo Markers</h1>
 Ola, <b>mundo!</b>"
```

2.2.4.1 INTERPRETADOR DE MARKDOWN

Adicionaremos um interpretador de Markers para Markdown. Começaremos definindo a seguinte função em `Markers.hs` :

```
convertToMarkdown :: String -> String
convertToMarkdown = parseFileWith toMarkdown
```

Também criaremos a função `toMarkdown` que será responsável por converter o tipo de dado `Markers` para uma string Markdown. A função `toMarkdown` será definida no módulo `Converter/To.hs` . Algo para se notar é que muitas das tags presentes no Markers não possuem conversão direta para Markdown. Sendo assim, a estilização de algumas tags não será possível, e o interpretador irá retornar apenas o texto puro.

Podemos definir a função `toMarkdown` da seguinte maneira:

```
toMarkdown :: Markers -> String
toMarkdown (MarkersMain titulo sections) = "# " <> titulo <> "\n\n"
           <> Prelude.foldr (\x acc -> helper x <> acc) "" sections
where
    helper :: MainSection -> String
    helper (Paragraph (Default content)) = content
    helper (Paragraph (Bold content)) = "**" <> content <> "**"
    helper (Paragraph (Italic content)) = "*" <> content <> "*"
    helper (Separator) = "---"
    helper (Paragraph (BoldItalic content)) = "***" <> content <> "***"

    -- Underline não é suportado em Markdown
    helper (Paragraph (Underlined content)) = "**" <> content <> "**"
    helper (Paragraph (Crossed content)) = "~~" <> content <> "~~"
    helper (Paragraph (CodeInline content)) = "`" <> content <> "`"

    -- Cor não é suportado em Markdown
    helper (Paragraph (Color _ content))  = "*" <> content <> "*"

    -- Convertendo p/ um header em negrito
    helper (Summary content) = "**" <> content <> "**"
    helper (Commentary content) = "<!-- " <> content <> " -->"
    -- Resto das conversões...
```

Podemos observar que a função `toMarkdown` é semelhante à função `toRaw` , porém com algumas diferenças. A principal diferença é que o Markdown não possui tags HTML, e sim uma sintaxe própria. Portanto, devemos utilizar a sintaxe do Markdown para cada tag. Por

exemplo, a tag `(b)` é convertida para `**bold**`, enquanto a tag `(i)` é convertida para `*italic*`.

Podemos testar nossa função `convertToMarkdown` chamando-a com o seguinte conteúdo:

```
convertToMarkdown "(title)Isso é um arquivo Markers(/title)
                   Ola, (b)mundo!(/b)"
```

Isto irá retornar a string Markdown:

```
"# Isso \233 um arquivo Markers\n\n
 Ola, **mundo!**"
```

Pelo fato de Markdown também ser uma linguagem de marcação simples que é interpretada para HTML, isso significa que o `convertToMarkdown` pode ser usado para uma instância de um *transpilador*; convertendo o arquivo Markers para o formato Markdown, e depois convertendo o Markdown para HTML em uma etapa futura - sendo assim, de uma linguagem de alto nível para outra de alto nível, que é depois convertida para uma linguagem de "baixo nível", neste caso, o HTML.

2.2.4.2 INTERPRETADOR PARA O FORMATO ESTILIZADO

Podemos levar vantagem do fato de estarmos interpretando nosso documento Markers para HTML, e utilizarmos o CSS para estilizar nosso documento. Podemos criar um interpretador que irá gerar um arquivo HTML estilizado. O interpretador irá gerar um arquivo HTML com o conteúdo lido, juntamente à uma tag `style` com o CSS necessário para estilizar o documento.

Iremos fazer isso primeiramente na função `toHtml`. O objetivo desta função será gerar este arquivo estilizado, dedicado à uma documentação polida e bem construida. Sua inspiração veio diretamente do layout de blocos da plataforma Notion. Começaremos importando alguns itens de significância, como o `htmx` para permitir chamadas APIs futuramente. Também importaremos a fonte `Fira Code` para seu uso em notações de código, como o bloco `code`. Definiremos também o título do documento na tag `title` do nosso HTML.

```
toHtml :: Markers -> String
toHtml (MarkersMain title sections) =
    "<!DOCTYPE html>\
    \<html lang=\"en\">\
```

```
\<head>\
\  <script src=\"https://unpkg.com/htmx.org@2.0.4\"></script>\
\  <meta charset=\"UTF-8\">\
\  <meta name=\"viewport\" content=\"width=device-width,
initial-scale=1.0\">\
\  <link \
\href=\"https://fonts.googleapis.com/css2?family=Fira+Code\"\
\ rel=\"stylesheet\">\
\  <title>" <> title <> "</title>\
```

Note os `\` que antecedem e sucedem cada linha. Isso é utilizado para que o Haskell não quebre a linha, e mantenha o código em uma única string. O `<>` é utilizado para concatenar strings de uma maneira geral, e o `\\` é utilizado para quebrar a linha. Também podemos utilizar da quebra de linha `\n`.

Após a definição do `header`, podemos definir a tag `style` com toda a estilização que queremos dar para nosso documento. O mesmo pode ser feito se referindo as próprias tags HTML, ou utilizando as classes. Observe o exemplo:

```
\  <style>\
\    body {\
\      margin: 0;\
\      padding: 0;\
\      line-height: 1.6;\
\      color: #333;\
\      text-align: justify;\
\      text-justify: inter-word;\
\    }\
\    .container {\
\      max-width: 800px;\
\      margin: 2em auto;\
\      padding: 2em;\
\      background-color: #fff;\
\      border-radius: 8px;\
\    }\
\    .container h1, .container h2, .container h3 {\
\      text-align: left;\
\      margin-top: 1.2em;\
\      margin-bottom: 0.8em;\
\    }\
\    .container p {\
\      line-height: 1.6;\
\      text-align: justify;\
\      margin: 1em 0;\
\    }\
-- Outras estilizações...\
```

Após finalizarmos a estilização, notamos que não estamos limitados a apenas o CSS, mas também podemos adicionar JavaScript para melhorar a experiência do usuário. Um dos melhores exemplos disso é a tag `(summary)` do Markers. A mesma não possúi capacidade na

AST para listar todos os capítulos presentes no documento; sendo assim, ela serve como uma "marcação" para uma `div` que será estilizada com o CSS e populada em tempo real via JavaScript. Adicionaremos os seguintes scripts na tag `head`:

```
// Adiciona o toggle das listas do Markers.

document.addEventListener('DOMContentLoaded', () => {
    const allDetails = document.querySelectorAll('details');
    allDetails.forEach(det => {
        det.addEventListener('toggle', () => {});
    });
});

// Verifica se um capítulo está aninhado, se estiver,
// diminui seu tamanho
// caso esteja dentro de outro capítulo.

document.querySelectorAll('.chapter h2').forEach(h2 => {
    const level = h2.closest('.chapter').parentElement?
    .closest('.chapter') ? 2 : 1;
    h2.style.fontSize = level === 2 ? '1.2em' : '1.5em';
});

// Popula e constrói o sumário com base em todas as
// tags de capítulo no doc
// usando Summary como uma marcação.

const summaryDiv = document.querySelector('.summary');
const chapters = document.querySelectorAll('.chapter h2');
const ul = document.createElement('ul');
chapters.forEach(chapter => {
    const li = document.createElement('li');
    li.textContent = chapter.textContent;
    ul.appendChild(li);
});

try { summaryDiv.appendChild(ul); } catch {}
```

Podemos então prosseguir normalmente com a construção do HTML, adicionando o conteúdo lido. Isto será feito na `div` container, dentro do `body`.

```
\<div class=\"container\">\
\<h1>" <> title <> "</h1>"
<> Prelude.foldr (\x acc -> helper x <> acc) "" sections <>
"</div>\
```

Com isso, permitimos uma melhor flexibilidade na exportação de documentos, sua utilização e funcionalidade. Esta habilidade e multiuso da linguagem é o que permite os documentos do Markers serem utilizados em diversos contextos, como por exemplo, para a construção de um site, ou para a construção de um documento técnico. O mesmo pode ser

utilizado para a construção de um livro, ou até mesmo para a construção de um artigo científico.

2.2.4.3 INTERPRETADOR PARA ABNT

O interpretador para a norma ABNT foi construido com o intuito de gerar um documento que siga as normas da ABNT perfeitamente, dado todas as definições de formatação e estrutura do documento com baixa (ou nula) margem para erro. Para isso ser possível, utilizamos das técnicas demonstradas anteriormente, e aplicamos a formatação necessária para que o documento siga as normas da ABNT. O interpretador irá gerar um arquivo HTML com o conteúdo lido, juntamente à uma tag `style` com o CSS necessário para estilizar o documento, além dos scripts JavaScript. Entraremos em detalhe sobre como a formatação da norma ABNT do Markers ocorre.

Este formato de arquivo foi idealizada juntamente a própria linguagem, isto é notável observando tags como `(ref)`. Uma das dificuldades encontradas na construção deste interpretador foram certas formatações (tal como paginação) que não são possíveis de serem feitas em HTML e CSS puro, para isso, também utlizamos de JavaScript.

Começaremos definindo três funções que usaremos durante a interpretação do documento, sendo estes `escapeHtml`, `splitSections`, `isSummary` e `stripPTags`. Estas funções são definidas da seguinte forma:

```
escapeHtml :: Char -> String
escapeHtml c = case c of
    '\n' -> "<br>"
    '<' -> "&lt;"
    '>' -> "&gt;"
    '&' -> "&amp;"
    '"' -> "&quot;"
    '\'' -> "&#39;"
    _ -> [c]

isSummary :: MainSection -> Bool
isSummary (Summary _) = True
isSummary _           = False

splitSections :: [MainSection] -> ([MainSection], [MainSection])
splitSections [] = ([], [])
splitSections xs =
let (pre, post) = break isSummary xs
in case post of
    []     -> (pre, [])
    (s:ss) -> (pre ++ [s], ss)
```

A função `escapeHtml` é responsável por "escapar" qualquer caractere HTML sensível que possa quebrar a estrutura do documento. Esta função realiza o pattern matching em cada possível caractere e o troca por um equivalente que não cause problemas ao ser interpretado.

A função `isSummary` verifica se algo do tipo Mainsection é ou não do tipo Summary, ela irá retornar um booleano com base em cada caso.

A função `splitSections` divide uma lista de MainSection em duas partes. A primeira parte contém todos os elementos até (e incluindo) a primeira ocorrência da tag `Summary`, enquanto a segunda parte contém os elementos restantes. Esta função será importante para a paginação do documento usando as regras de CSS, já que devemos iniciar a contagem da primeira página, porém só podemos mostrar a paginação depois do sumário, que sempre será o último elemento de "cabeçalho" antes do conteúdo.

Iniciaremos definindo as margens e estrutura da página via CSS.

```
toAbnt :: Markers -> String
toAbnt (MarkersMain someString sections) =
  -- Divide as seções antes e depois do sumário.
  let (preSections, postSections) = splitSections sections in
    "<!DOCTYPE html>\n\
    \<html lang=\"pt-BR\">\n\
    \<head>\n\
    \  <meta charset=\"UTF-8\">\n\
    \  <title>" <> someString <> "</title>\n\
    \  <style>\n\
    \    @page {\n\
    \      size: A4;\n\
    \      margin: 3cm 2cm 2cm 3cm;\n\
    \\
    \      /* Cabeçalho padrão com numeração */\n\
    \\
    \      @top-right {\n\
    \        content: counter(page, decimal);\n\
    \        font-family: 'Times New Roman', Times, serif;\n\
    \        font-size: 12pt;\n\
    \      }\n\
    \    }\n\
    \    @page noheader {\n\
    \      @top-right { content: \"\"; }\n\
    \    }\n\
    \\
    \\
    \    .pre-summary { page: noheader; }\n\
    \    body {\n\
    \      font-family: 'Times New Roman', Times, serif;\n\
    \      font-size: 12pt;\n\
    \      line-height: 1.5;\n\
    \      text-align: justify;\n\
    \      color: #000;\n\
```

```
\      margin: 0;\n\
\      padding: 0;\n\
\    }\n\
\\ -- Outras estilizações específicas da ABNT...
```

Após definirmos todo o estilo do documento, podemos inserir os scripts necessários para a formatação correta do documento. Iremos começar inserindo os scripts necessários para seguirmos a norma ABNT corretamente.

```
<script>

// Quando o DOM carregar por completo, procure o elemento
// "summary". Caso ele exista, crie um elemento ul
// para cada capítulo.

document.addEventListener('DOMContentLoaded', () => {
    const summaryDiv = document.querySelector('.summary');
    if (!summaryDiv) return;
    // criamos o primeiro UL e o contador
    let ul = document.createElement('ul');
    let itemCount = 0;
```

Após isso, prosseguimos com a construção do sumário. O sumário será construído com base em todos os capítulos presentes no documento. Para isso, utilizamos a função `generateChapterList` que irá gerar o sumário de forma recursiva, contando os capítulos e subcapítulos presentes no documento. Esta função é definida da seguinte forma; Primeiramente recebemos uma lista de elementos `h2` que são os capítulos do documento. Para cada capítulo, calculamos o número do capítulo e o título seu título. Criamos um elemento `li` para cada um deles.

Também procuramos por subcapítulos, e para cada um deles chamamos a função recursiva para gerar o número do subcapítulo e o título. Forçamos a quebra de página a cada 23 capítulos (ou seja, a cada 24 itens). Este é o limite de capítulos presentes da Norma ABNT.

```
function generateChapterList(chapters, parentNumber = '') {
    chapters.forEach((chapter, index) => {
    const currentNumber = parentNumber
    ? `${parentNumber}.${index + 1}`
    : `${index + 1}`;
    const h2 = chapter.querySelector(':scope > h2');
    if (!h2) return;
    const chapterTitle = h2.textContent.trim();
    h2.innerHTML = `<span class="chapter-number">
        ${currentNumber}</span> ${chapterTitle}`;

    const li = document.createElement('li');
    const pageElem = chapter.querySelector('#chapterPageNumber');
```

```
        const pageNumber = pageElem ? pageElem.textContent.trim() : "";
        li.innerHTML = `<span class="chapter-number">
        ${currentNumber}</span>
            <span class="chapter-title">${chapterTitle}</span>
            <span class="dots"></span> <span class="page">
            ${pageNumber}</span>`;
        ul.appendChild(li);

        // Fallback simples: a cada 24 itens, força quebra de página
        if ((index + 1) % 23 === 0) {
        li.style.pageBreakAfter = 'always';
        }

        const nested = chapter.querySelectorAll(':scope > .chapter');
        if (nested.length > 0) {
        generateChapterList(Array.from(nested), currentNumber);
        }
        });
    }
```

Por fim, selecionaremos todos os elementos com a classe `.chapter` e os converteremos em um array. Iremos filtrar este array para manter somente os capítulos que não estão aninhados com outros elementos da classe `.chapter`. Para cada um deles, chamamos a função `generateChapterList` para gerar o sumário. Verificamos então se este último elemento `ul` possui filhos, e se sim, o anexamos ao elemento `summaryDiv`. O código final do script JavaScript é o seguinte:

```
const topChapters = Array.from(document.querySelectorAll('.chapter'))
      .filter(c => !c.parentElement.closest('.chapter'));
      generateChapterList(topChapters);
      if (ul.children.length) summaryDiv.appendChild(ul);
});
```

Podemos usar a mesma lógica desta função para popular outras listas, como as listas de figuras.

Outro desafio que tivemos estava relacionado com a identação de cada parágrafo. Em nossa função `helper`, definimos o *pattern matching* de `Paragraph (Default content)` da seguinte maneira:

```
helper :: MainSection -> String
helper (Paragraph (Default content)) =
  "<p class=\"indent\">" <> content <> "</p>"
```

Isso introduz um bug onde tags de formatação de texto sempre apareciam em uma nova linha, mesmo que estivessem dentro de um parágrafo. Criando assim identações aninhadas, o que não foi desejado.

A solução para este problema diretamente no parser e no interpretador provou ser complicada demais para implementar de forma concisa, foi então decidido que a solução se daria pela adição de um novo script JavaScript que irá remover as tags de parágrafo que estão dentro das tags que foram convertidas, sendo estes `code`, `strong`, `em` e `span`.

```
document.addEventListener('DOMContentLoaded', () => {
  document.querySelectorAll('strong, em, span, code')
  .forEach(inline => {
      const prev = inline.previousElementSibling;
      const next = inline.nextElementSibling;
      if (
          prev && next &&
          prev.tagName === 'P' &&
          prev.classList.contains('indent') &&
          next.tagName === 'P' &&
          next.classList.contains('indent')
      ) {
          const combined = prev.innerHTML + inline.outerHTML
          + next.innerHTML;
          const merged = document.createElement('p');
          merged.className = 'indent';
          merged.innerHTML = combined;
          next.remove();
          inline.remove();
          prev.replaceWith(merged);
      }
  });
);
```

Usaremos a mesma lógica que usamos anteriormente no sumário para gerar as referências com base nas tags `ref`. Usaremos uma marcação de onde as referências aparecerão baseada no tipo de dado `References`. Sua interpretação será a seguinte:

```
helper References = "<div class=\"references\">"
                    <> "<h2 class=\"summary-title\">"
                    <> "REFERÊNCIAS"
                    <> <div class=\"references-list\">
                    <> "</div>"
```

As referências neste formato de exportação são determinadas por um parágrafo completo cujo possúi atributos invisiveis em tags `span`.

```
helper (Ref url author title year access content) =
  "<span class=\"reference\">" <>
  "<span style=\"visibility: none; class=\"title\">"
  <> title <> "</span>" <>
  "<span style=\"visibility: none; class=\"author\">"
  <> author <> "</span>" <>
  "<span style=\"visibility: none; class=\"year\">"
  <> year <> "</span>" <>
```

```
      "<span style=\"visibility: none; class=\"access\">"
      <> access <> "</span>" <>
      "<span style=\"visibility: none; class=\"url\">"
      <> url <> "</span>" <>
      "<span class=\"content\" style=\"display:inline;\">"
      <> foldr (\x acc -> helper x <> acc) "" content
      <> "</span>"
      <> "</span>"
```

Estes dados serão pegos um por um por nossa função JavaScript e serão adicionados a uma lista de referências (aqui sendo `references-list`). Esta função irá selecionar todos os elementos contidos em `post-summary` que possuam a classe `span.reference`. Para cada referência encontrada, será extraido os textos individuais que se encontram invisíveis. Será criado então um elemento de parágrafo para cada referência, e o mesmo será adicionado a lista de referências. O código JavaScript para isso é o seguinte:

```
function populateABNTReferences() {
    const list = document.querySelector('.post-summary
    .references-list');
    if (!list || list.children.length) return;

    const refs = document.querySelectorAll('.post-summary
    span.reference');
    refs.forEach(ref => {
      const author = ref.querySelector('.author')
       ?.textContent.trim() || '';
      const title  = ref.querySelector('.title')
       ?.textContent.trim()  || '';
      const year   = ref.querySelector('.year')
       ?.textContent.trim()   || '';
      const url    = ref.querySelector('.url')
       ?.textContent.trim()    || '';
      const access = ref.querySelector('.access')
       ?.textContent.trim() || '';

      const p = document.createElement('p');
     p.style.textAlign = 'left';
     p.style.lineHeight = '1';
     p.innerHTML =
        `${author}. <b>${title.toUpperCase()}</b>, ${year}.
        Disponível em: ` +
        `<<a style=\"color: black; text-decoration: none;\"
            href=\\\"${url}\\\" target=\\\"_blank\\\">${url}</a>>.`
        + `Acesso em: ${access}.`;

      list.appendChild(p);
    });
  }

  document.addEventListener('vivliostyle-rendering-completed',
  populateABNTReferences);
  document.addEventListener('DOMContentLoaded',
  populateABNTReferences);
```

Neste tipo de formatação, o uso da tag `(meta)` é estritamente necessário. Ela irá definir o conteúdo da capa e contracapa do documento. Faremos isso utilizando de algumas funções auxiliares de caixa preta dentro da função `helper`. Para definirmos a capa em seu formato respectivo à norma ABNT, utilizaremos da função `helperTopAbnt` - que irá definir o estilo do topo da capa, e `helperBottomAbnt`, que irá definir o rodapé do mesmo. Estas funções usaram do *pattern matching* no tipo de dado `MetaSection` que está definido dentro de `MainSection`.

```
helperTopAbnt :: MetaSection -> String
helperTopAbnt (Institution c) =
  "<div style=\"text-align: center;\">"
  <> <p class=\"institution\" style=\"margin-bottom: 30%\"<>><b>"
  <> concatMap escapeHtml c <> "</b></p></div>"

helperTopAbnt (Author c) =
  "<div style=\"text-align: center;\">"
  <> "<p class=\"author\" style=\"margin-bottom: 30%\">"
  <> c <> "</p></div>"

helperTopAbnt _ = ""


helperBottomAbnt :: MetaSection -> String
helperBottomAbnt (Subtitle c) =
  "<div style=\"text-align: center;\">"
  <> <p class=\"subtitle\" style=\"margin-top: -15px;
  margin-bottom: 55%\">" <> c <> "</p></div>"

helperBottomAbnt (Location c) =
  "<div style=\"text-align: center;\">"
  <> <p class=\"location\">" <> c <> "</p></div>"
helperBottomAbnt (Year c) =
  "<div style=\"text-align: center;\">"
  <> <p class=\"year\" style=\"margin-bottom: 80px\">"
  <> c <> "</p></div>"
helperBottomAbnt _ = ""
```

Utilizando das margens e estilos CSS especificados, podemos criar uma réplica exata de uma capa e contracapa da ABNT, renderizando o conteúdo vindo direto da tag `(meta)`.

FIGURA 9 – Exportação da capa da norma ABNT.

CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
Faculdade de Tecnologia Baixada Santista Rubens Lara
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO
UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

SANTOS, SP
2025

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO
UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia Rubens Lara, como exigência para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

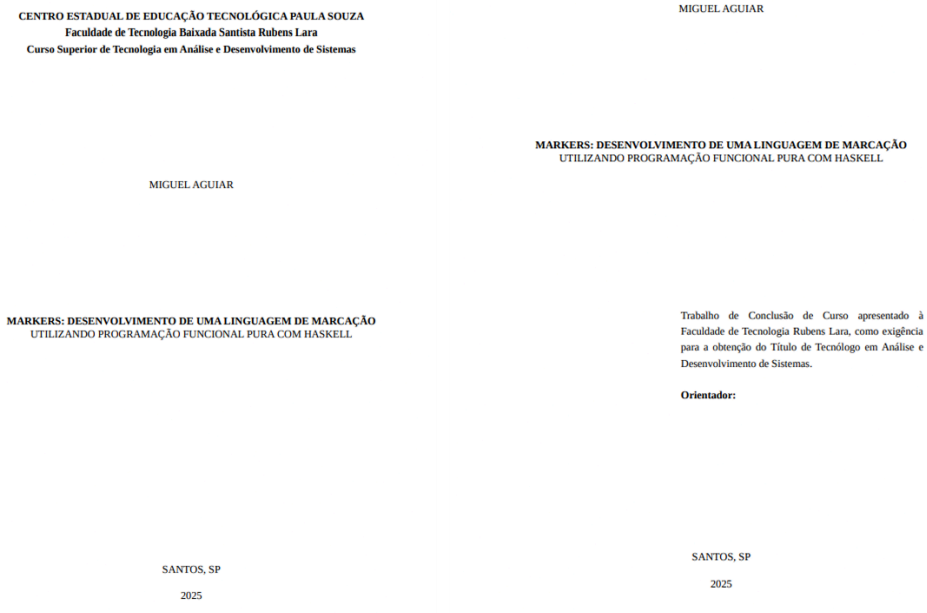
Orientador:

SANTOS, SP
2025

Fonte: Autor, 2025

Notemos também a geração automática do sumário e lista de figuras.

FIGURA 10 – Exportação do sumário e lista de figuras.



CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
Faculdade de Tecnologia Baixada Santista Rubens Lara
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO
UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

SANTOS, SP
2025

MIGUEL AGUIAR

MARKERS: DESENVOLVIMENTO DE UMA LINGUAGEM DE MARCAÇÃO
UTILIZANDO PROGRAMAÇÃO FUNCIONAL PURA COM HASKELL

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia Rubens Lara, como exigência para a obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

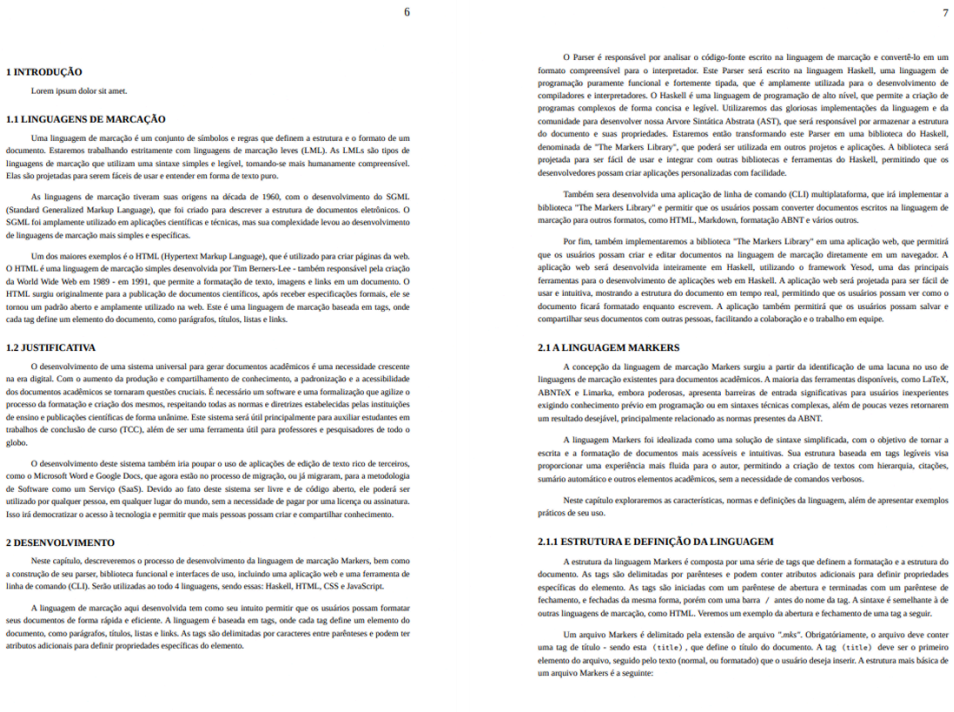
Orientador:

SANTOS, SP
2025

Fonte: Autor, 2025

Da estilização do conteúdo, com paginação automática após o sumário;

FIGURA 11 – Exportação do conteúdo da norma ABNT.



6

1 INTRODUÇÃO

Lorem ipsum dolor sit amet.

1.1 LINGUAGENS DE MARCAÇÃO

Uma linguagem de marcação é um conjunto de símbolos e regras que definem a estrutura e o formato de um documento. Estaremos trabalhando estritamente com linguagens de marcação leves (LML). As LMLs são tipos de linguagens de marcação que utilizam uma sintaxe simples e legível, tornando-se mais humanamente compreensível. Elas são projetadas para serem fáceis de usar e entender em forma de texto puro.

As linguagens de marcação tiveram suas origens na década de 1960, com o desenvolvimento do SGML (Standard Generalized Markup Language), que foi criado para descrever a estrutura de documentos eletrônicos. O SGML foi amplamente utilizado em aplicações científicas e técnicas, mas sua complexidade levou ao desenvolvimento de linguagens de marcação mais simples e específicas.

Um dos maiores exemplos é o HTML (Hypertext Markup Language), que é utilizado para criar páginas da web. O HTML é uma linguagem de marcação simples desenvolvida por Tim Berners-Lee - também responsável pela criação da World Wide Web em 1989 - em 1991, que permite a formatação de texto, imagens e links em um documento. O HTML surgiu originalmente para a publicação de documentos científicos, após receber especificações formais, ele se tornou um padrão aberto e amplamente utilizado na web. Este é uma linguagem de marcação baseada em tags, onde cada tag define um elemento do documento, como parágrafos, títulos, listas e links.

1.2 JUSTIFICATIVA

O desenvolvimento de uma sistema universal para gerar documentos acadêmicos é uma necessidade crescente na era digital. Com o aumento da produção e compartilhamento de conhecimento, a padronização e a acessibilidade dos documentos acadêmicos se tornaram questões cruciais. É necessário um software e uma formalização que agilize o processo da formatação e criação dos mesmos, respeitando todas as normas e diretrizes estabelecidas pelas instituições de ensino e publicações científicas de forma unânime. Este sistema será útil principalmente para auxiliar estudantes em trabalhos de conclusão de curso (TCC), além de ser uma ferramenta útil para professores e pesquisadores de todo o globo.

O desenvolvimento deste sistema também iria poupar o uso de aplicações de edição de texto rico de terceiros, como o Microsoft Word e Google Docs, que agora estão no processo de migração, ou já migraram, para a metodologia de Software como um Serviço (SaaS). Devido ao fato deste sistema ser livre e de código aberto, ele poderá ser utilizado por qualquer pessoa, em qualquer lugar do mundo, sem a necessidade de pagar por uma licença ou assinatura. Isso irá democratizar o acesso à tecnologia e permitir que mais pessoas possam criar e compartilhar conhecimento.

2 DESENVOLVIMENTO

Neste capítulo, descreveremos o processo de desenvolvimento da linguagem de marcação Markers, bem como a construção de seu parser, biblioteca funcional e interfaces de uso, incluindo uma aplicação web e uma ferramenta de linha de comando (CLI). Serão utilizadas ao todo 4 linguagens, sendo essas: Haskell, HTML, CSS e JavaScript.

A linguagem de marcação aqui desenvolvida tem como seu intuito permitir que os usuários possam formatar seus documentos de forma rápida e eficiente. A linguagem é baseada em tags, onde cada tag define um elemento do documento, como parágrafos, títulos, listas e links. As tags são delimitadas por caracteres entre parênteses e podem ter atributos adicionais para definir propriedades específicas do elemento.

7

O Parser é responsável por analisar o código-fonte escrito na linguagem de marcação e convertê-lo em um formato compreensível para o interpretador. Este Parser será escrito na linguagem Haskell, uma linguagem de programação puramente funcional e fortemente tipada, que é amplamente utilizada para o desenvolvimento de compiladores e interpretadores. O Haskell é uma linguagem de programação de alto nível, que permite a criação de programas complexos de forma concisa e legível. Utilizaremos das gloriosas implementações da linguagem e da comunidade para desenvolver nossa Arvore Sintática Abstrata (AST), que será responsável por armazenar a estrutura do documento e suas propriedades. Estaremos então transformando este Parser em uma biblioteca do Haskell, denominada de "The Markers Library", que poderá ser utilizada em outros projetos e aplicações. A biblioteca será projetada para ser fácil de usar e integrar com outras bibliotecas e ferramentas do Haskell, permitindo que os desenvolvedores possam criar aplicações personalizadas com facilidade.

Também sera desenvolvida uma aplicação de linha de comando (CLI) multiplataforma, que irá implementar a biblioteca "The Markers Library" e permitir que os usuários possam converter documentos escritos na linguagem de marcação para outros formatos, como HTML, Markdown, formatação ABNT e vários outros.

Por fim, também implementaremos a biblioteca "The Markers Library" em uma aplicação web, que permitirá que os usuários possam criar e editar documentos na linguagem de marcação diretamente em um navegador. A aplicação web será desenvolvida inteiramente em Haskell, utilizando o framework Yesod, uma das principais ferramentas para o desenvolvimento de aplicações web em Haskell. A aplicação web será projetada para ser fácil de usar e intuitiva, mostrando a estrutura do documento em tempo real, permitindo que os usuários possam ver como o documento ficará formatado enquanto escrevem. A aplicação também permitirá que os usuários possam salvar e compartilhar seus documentos com outras pessoas, facilitando a colaboração e o trabalho em equipe.

2.1 A LINGUAGEM MARKERS

A concepção da linguagem de marcação Markers surgiu a partir da identificação de uma lacuna no uso de linguagens de marcação existentes para documentos acadêmicos. A maioria das ferramentas disponíveis, como LaTeX, ABNTeX e Limarka, embora poderosas, apresenta barreiras de entrada significativas para usuários inexperientes exigindo conhecimento prévio em programação ou em sintaxes técnicas complexas, além de poucas vezes retornarem um resultado desejável, principalmente relacionado as normas presentes da ABNT.

A linguagem Markers foi idealizada como uma solução de sintaxe simplificada, com o objetivo de tornar a escrita e a formatação de documentos mais acessíveis e intuitivas. Sua estrutura baseada em tags legíveis visa proporcionar uma experiência mais fluida para o autor, permitindo a criação de textos com hierarquia, citações, sumário automático e outros elementos acadêmicos, sem a necessidade de comandos verbosos.

Neste capítulo exploraremos as características, normas e definições da linguagem, além de apresentar exemplos práticos de seu uso.

2.1.1 ESTRUTURA E DEFINIÇÃO DA LINGUAGEM

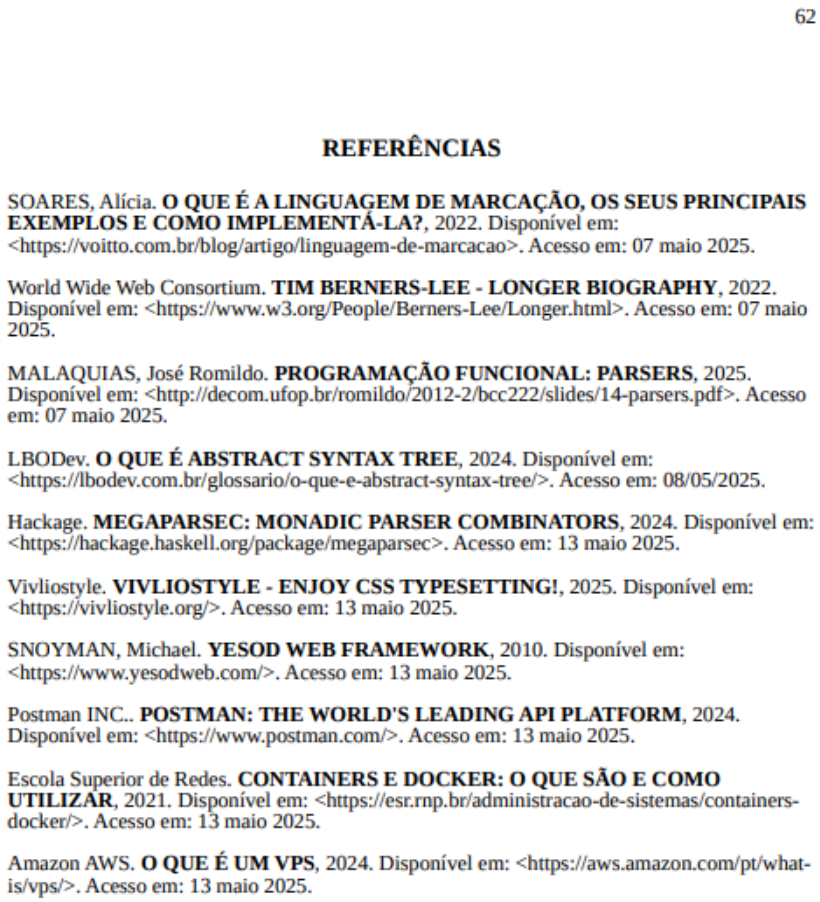
A estrutura da linguagem Markers é composta por uma série de tags que definem a formatação e a estrutura do documento. As tags são delimitadas por parênteses e podem conter atributos adicionais para definir propriedades específicas do elemento. As tags são iniciadas com um parêntese de abertura e terminadas com um parêntese de fechamento, e fechadas da mesma forma, porém com uma barra `/` antes do nome da tag. A sintaxe é semelhante à de outras linguagens de marcação, como HTML. Veremos um exemplo da abertura e fechamento de uma tag a seguir.

Um arquivo Markers é delimitado pela extensão de arquivo *".mks"*. Obrigatóriamente, o arquivo deve conter uma tag de título - sendo esta `(title)`, que define o título do documento. A tag `(title)` deve ser o primeiro elemento do arquivo, seguido pelo texto (normal, ou formatado) que o usuário deseja inserir. A estrutura mais básica de um arquivo Markers é a seguinte:

Fonte: Autor, 2025

E da criação automática da página de referências com base nas tags `(ref)`:

FIGURA 12 – Exportação das referências da norma ABNT.



62

REFERÊNCIAS

SOARES, Alícia. **O QUE É A LINGUAGEM DE MARCAÇÃO, OS SEUS PRINCIPAIS EXEMPLOS E COMO IMPLEMENTÁ-LA?**, 2022. Disponível em: <https://voitto.com.br/blog/artigo/linguagem-de-marcacao>. Acesso em: 07 maio 2025.

World Wide Web Consortium. **TIM BERNERS-LEE - LONGER BIOGRAPHY**, 2022. Disponível em: <https://www.w3.org/People/Berners-Lee/Longer.html>. Acesso em: 07 maio 2025.

MALAQUIAS, José Romildo. **PROGRAMAÇÃO FUNCIONAL: PARSERS**, 2025. Disponível em: <http://decom.ufop.br/romildo/2012-2/bcc222/slides/14-parsers.pdf>. Acesso em: 07 maio 2025.

LBODev. **O QUE É ABSTRACT SYNTAX TREE**, 2024. Disponível em: <https://lbodev.com.br/glossario/o-que-e-abstract-syntax-tree/>. Acesso em: 08/05/2025.

Hackage. **MEGAPARSEC: MONADIC PARSER COMBINATORS**, 2024. Disponível em: <https://hackage.haskell.org/package/megaparsec>. Acesso em: 13 maio 2025.

Vivliostyle. **VIVLIOSTYLE - ENJOY CSS TYPESETTING!**, 2025. Disponível em: <https://vivliostyle.org/>. Acesso em: 13 maio 2025.

SNOYMAN, Michael. **YESOD WEB FRAMEWORK**, 2010. Disponível em: <https://www.yesodweb.com/>. Acesso em: 13 maio 2025.

Postman INC.. **POSTMAN: THE WORLD'S LEADING API PLATFORM**, 2024. Disponível em: <https://www.postman.com/>. Acesso em: 13 maio 2025.

Escola Superior de Redes. **CONTAINERS E DOCKER: O QUE SÃO E COMO UTILIZAR**, 2021. Disponível em: <https://esr.rnp.br/administracao-de-sistemas/containers-docker/>. Acesso em: 13 maio 2025.

Amazon AWS. **O QUE É UM VPS**, 2024. Disponível em: <https://aws.amazon.com/pt/what-is/vps/>. Acesso em: 13 maio 2025.

Fonte: Autor, 2025

Após finalizar os processos de estilização, suporte das tags e inserção de scripts, podemos notar que sua exportação se equipara perfeitamente com todas as definições da

Associação Brasileira de Normas Técnicas a norma NBR 14724, responsável por monografias e trabalhos acadêmicos.

Observaremos os resultados obtidos em casos de uso do Markers para desenvolvimento de artigos e trabalhos acadêmicos durante o capítulo 2.6.

2.2.5 A LINGUAGEM EULER

A linguagem *Euler*, é uma sub-linguagem de marcação para fins de demonstrações matemáticas na linguagem Markers. Ele diverge da sintáxe focada em simplicidade do Markers; optando por oferecer uma alternativa mais verbosa para representação de operações matemáticas, isto foi idealizado principalmente devido a possibilidade de usuários da linguagem poderem não ter um apoio matemático firme. A linguagem é separada em quatro categorias de construtores, sendo estes: Termos Atômicos, operadores, agrupamentos e funções especiais.

Termos atômicos são os blocos básicos dentro de uma tag `(math)`: representam números inteiros, variáveis simbólicas ou reticências para indicar continuação. Exemplos típicos incluem o algarismo `1`, a variável `x_1` ou as reticências `…`, que, internamente, são mapeadas para os construtores `Number`, `Var` e `Ellipsis` do parser.

Os operadores definem como dois ou mais termos se combinam para formar expressões compostas. Na categoria de operadores, possuimos três variantes: Unários (tal como a negação em `-2`), Binários aritméticos (tal como soma, subtração, multiplicação...) e Binários relacionais - utilizados para representar igualdade com `=` e *ponteiros*, tal como em `[lim | x to 0]`.

Os agrupamentos controlam a ordem de avaliação. Utilizamos de parenteses em subexpressões para forçarmos sua resolução antecipada. Também introduzimos o uso de colchetes, utilizado para a construção de funções especiais. Estas funções especiais oferecem sintaxes de alto nível para construções matemáticas frequentes. As funções são invocadas por seu nome como primeiro atributo da abertura da função, seguido dos parâmetros necessários. Caso queiremos representar `a + b` dividido por `c`, podemos utilizar do seguinte código:

```
(math)
[fraction | (a + b) | c]
(/math)
```

Que irá ser convertido para a representação visual de:

$$(a \ + \ \frac{}{c} \ b)$$

Utilizando uma fórmula de regressão logística como exemplo, podemos observar que o aninhamento destas funções em Euler é possível, observe o uso da função `power-of` como parte do segundo argumento de `fraction` .

```
(math)
[prob | y = 1 | x] = [fraction | 1 | 1 +
    [power-of | e |
    - (beta_0 + beta_1 + x_1 + ... + beta_p & x_p)
    ]
]
(/math)
```

Este código irá ser formatado então para:

$$P(y \ = \ 1 \mid x) \ = \ \frac{1}{1 \ + \ e^{-(\beta_0 \ + \ \beta_1 \ + \ x_1 \ + \ ... \ + \ \beta_p x_p)}}$$

A implementação da linguagem Euler será dada pela criação de uma nova AST, chamaremos este de `EulerSyntaxTree.hs` . Este será responsável pelos tipos de dados das operações e variadas funções da linguagem.

```
data MathExpr
= Number String
| Add MathExpr MathExpr
| Sub MathExpr MathExpr
| Mul MathExpr MathExpr
| ImplicitMul MathExpr MathExpr
| Div MathExpr MathExpr
| PowerOf    MathExpr MathExpr
| SquareRoot MathExpr
| Fraction   MathExpr MathExpr
| Var String
| Eq         MathExpr MathExpr
| Arrow      MathExpr MathExpr
| Neg MathExpr
| Probability MathExpr MathExpr
| Ellipsis
| Parens     MathExpr
| Sum        MathExpr MathExpr MathExpr
| Product    MathExpr MathExpr MathExpr
| Integral   (Maybe (MathExpr,MathExpr)) MathExpr
| Limit      MathExpr MathExpr
| Derivative MathExpr MathExpr
```

```
    | Root       (Maybe MathExpr) MathExpr
    | Binom      MathExpr MathExpr
    | Abs        MathExpr
    | Vector     [MathExpr]
    | Matrix     [[MathExpr]]
    | Func       String [MathExpr]
    deriving (Show, Eq)
```

Podemos então prosseguir com a criação de nosso parser. Este irá contar com algumas mudanças em relação ao parser apresentado na linguagem Markers. Começaremos definindo uma tabela de operações. Para definirmos com sucesso, será necessário o das bibliotecas `parser-combinators`, `containers` e `split`.

```
operatorTable :: [[Operator Parser MathExpr]]
operatorTable =
[ [ Prefix  (Neg <$ symbol "-") ]
, [ InfixL  (ImplicitMul <$ symbol "&")
, InfixL  (Mul         <$ symbol "*")
, InfixL  (Div         <$ symbol "/") ]
, [ InfixL  (Add         <$ symbol "+")
, InfixL  (Sub <$ lexeme (string "-"
          <* notFollowedBy (char '>')))
]
, [ InfixN  (Arrow <$ symbol "->"
           <|> Arrow <$ symbol "to" )
, InfixN  ( Eq    <$ symbol "=" )
  ]
]
```

Podemos então utilizar `operatorTable` como parâmetro para construir o analisador de expressões propriamente dito, com todas as regras de precedência e associatividade já definidas. Para isso, a função necessária é a `makeExprParser`, vinda da biblioteca `Text.Megaparsec.Expr`:

```
lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

symbol :: String -> Parser String
symbol = L.symbol sc

parensP :: Parser MathExpr
parensP = Parens <$>
    between (symbol "(") (symbol ")") parseExpr

parseExpr :: Parser MathExpr
parseExpr = makeExprParser term operatorTable
```

Para a sintáxe específica das funções, utilizaremos de `parseFunction` . Esta função também será responsável por definir cada uma das palavras-chave que serão utilizadas dentro da sintáxe de colchetes. Uma versão resumida de `parseFunction` seria:

```
parseFunction :: Parser MathExpr
parseFunction = brackets $ do
  name <- lexeme $ choice $ Prelude.map string
    [ "fraction", "power-of", "square-root"
    , "prob", "sum" ]
  _    <- symbol "|"
  args <- parseExpr `sepBy1` symbol "|"
  case (name, args) of
    ("fraction",    [n, d]) ->
        pure (Fraction   n d)
    ("power-of",    [b, e]) ->
        pure (PowerOf    b e)
    ("square-root", [x    ]) ->
        pure (SquareRoot x)
    ("prob",        [evt, c]) ->
        pure (Probability evt c)
    ("sum",    [i0, iN, body])     ->
        pure (Sum     i0 iN body)
    _ -> fail $ "wrong argumentss for math func " ++ name
```

Após isso, também iremos definir a função `term` , sendo este o parser que reconhece termos atômicos, agrupamentos e funções especiais. Sua implementação é a seguinte:

```
term :: Parser MathExpr
term =
    try parseFunction
<|> try parseCall
<|> parensP
<|> Ellipsis
    <$  lexeme (string "..." <|> string "…")
<|> Var <$> lexeme ((:) <$> letterChar
    <*> many (letterChar <|> digitChar <|> char '_'))
<|> Number <$> lexeme (some digitChar)
```

Por fim, definimos o parser de bloco matemático completo, que identifica as tags de abertura e fechamento.

```
parseMathBlock :: Parser MainSection
parseMathBlock = MathBlock . pure
<$> between (symbol "(math)") (symbol "(/math)") parseExpr
```

Para a parte de interpretação, adicionaremos o bloco `MathBlock [MathExpr]` na AST da linguagem Markers, com isso, podemos realizar o *pattern matching* do ADT como visto anteriormente no capítulo 2.2.4.

```
helper (MathBlock expression) =
    "<div class=\"math-block\">"
    <> foldr (\x acc -> renderMath x <> acc) "" expression
    <> "</div>"
    where
    renderMath expr = case expr of
        Number n        -> n
        Add a b         -> renderBin a " + "  b
        Sub a b         -> renderBin a " − "  b
        Mul a b         -> renderBin a " · "  b
        Div a b         -> renderBin a " ÷ "  b
        -- (...)
        Var v ->
        let parts = splitOn "_" v
            base  = head parts
            subs  = tail parts
            sym0  = M.findWithDefault base base greekMap
        in  foldl (\acc sub -> acc ++ "<sub>" ++ sub ++ "</sub>")
                  sym0
                  subs
```

Podemos notar no *pattern matching* de Var a linha `M.findWithDefault base base greekMap` . Esta parte será responsável por mapear letras gregas aos seus nomes com base em uma tabela pré-pronta. Supondo que queiramos representar "λ de número 34" em uma notação matemática, podemos escrever `lambda_34` ; isto será detectado como variável, o `_` como representante para números subscritos. Sendo assim:

λ34

A lista para mapeamento das letras gregas é a seguinte:

```
greekMap :: M.Map String String
greekMap = M.fromList
  [ ("alpha",      "α"),   ("beta",       "β"),   ("gamma",  "γ")
  , ("delta",      "δ"),   ("epsilon",    "ε"),   ("zeta",   "ζ")
  , ("eta",        "η"),   ("theta",      "θ"),   ("iota",   "ι")
  , ("kappa",      "κ"),   ("lambda",     "λ"),   ("mu",     "μ")
  , ("nu",         "ν"),   ("xi",         "ξ"),   ("omicron","ο")
  , ("pi",         "π"),   ("rho",        "ρ"),   ("sigma",  "σ")
  , ("tau",        "τ"),   ("upsilon",    "υ"),   ("phi",    "φ")
  , ("chi",        "χ"),   ("psi",        "ψ"),   ("omega",  "ω")
  ]
```

2.2.6 ENVIANDO O PROJETO PARA O GITHUB

Após sua conclusão, enviaremos o que chamaremos de "The Markers Library" para o repositório do projeto. Será utilizado o Github para o envio do projeto. O repositório é público

e protegido por uma licença MIT, permitindo que qualquer um possa utilizar o projeto e contribuir com ele.

Utilizaremos da ferramenta `git` para enviar o projeto para o Github. O git é um sistema de controle de versão que permite que possamos gerenciar o código-fonte do projeto, permitindo que possamos voltar a versões anteriores do código, e também permitindo que possamos trabalhar em equipe de forma colaborativa.

Criaremos um repositório no Github com o nome "markers", e enviaremos o projeto para lá. O repositório será útil não apenas para versionamento, mas para seu uso nos demais projetos que utilizam o Markers como base. O repositório do The Markers Library está disponível em https://github.com/TheMarkersFoundation/markers (Acesso em: 13 maio 2025).

2.3 DESENVOLVIMENTO DO APLICATIVO CLI

Com a conclusão do The Markers Library, podemos prosseguir para o desenvolvimento do aplicativo CLI. O aplicativo CLI será responsável por receber o arquivo Markers e convertê-lo para o formato desejado. Utilizaremos do The Markers Library que fora disponibilizado no repositório do projeto para realizar o parsing e a conversão do arquivo. O aplicativo CLI será desenvolvido em Haskell, e será um súite simples que converterá um arquivo markers em um arquivo de formato desejado (HTML, ABNT, Markdown, etc) com o passar de um parâmetro.

Utilizaremos do `stack`, ferramenta do Haskell para gerenciamento de pacotes e construção de projetos, o stack permite com que usemos bibliotecas disponíveis em repositórios remotos, como no Github; ao invés de ser restrito ao Hackage como o cabal. O stack também permite que utilizemos versões específicas de bibliotecas, o que é útil para evitar problemas de compatibilidade entre versões. Ele também possui um sistema de gerenciamento de dependências, o que facilita a instalação e atualização de pacotes.

Usaremos as bibliotecas `directory`, `filepath` e `bytestring`, além da The Markers Library para o desenvolvimento do nosso aplicativo (doravante nomeado The Markers Parser). A biblioteca `directory` é utilizada para manipulação de arquivos e diretórios, a biblioteca `filepath` é utilizada para manipulação de caminhos de arquivos, e a biblioteca `bytestring` é utilizada para manipulação de strings em bytes. Estas bibliotecas são úteis para o desenvolvimento do aplicativo CLI, pois iremos precisar manipular arquivos e diretórios, além de manipular strings.

Iremos editar o arquivo `stack.yaml` e adicionar a seguinte linha na seção `extra-deps`:

```
extra-deps:
- github: TheMarkersFoundation/TheMarkersLibrary
    commit: 660d71bcf4bd5a8fbae909b26cc3e218cadd912e
```

Isto irá adicionar a biblioteca The Markers Library como dependência do projeto, sendo puxada diretamente do repositório do Github. Teremos um único arquivo chamado `Main.hs` dentro de um diretório chamado `src`. O Markers Parser é um aplicativo pequeno que necessitará apenas deste módulo. Iremos começar definindo a função `printHelp`:

```
printHelp :: IO ()
printHelp = putStrLn "Usage: mks [-html <file-name>"
        ++ " | -markdown <file-name>"
        ++ " | -md <file-name> "
        ++ " | -abnt <file-name> "
        ++ " | -help | -h]"
```

Esta função será usada como caso base de nosso *pattern matching* na função `main`. Ela irá imprimir na tela como utilizar o aplicativo, e quais opções estão disponíveis. A função `main` será definida da seguinte maneira:

```
main :: IO ()
main = do
    args <- getArgs
    case args of
        _ -> printHelp
```

Iremos definir cada uma das possibilidades de conversão, utilizando o *pattern matching* para verificar qual opção foi passada como parâmetro. A função `getArgs` é uma função da biblioteca `System.Environment` que retorna os argumentos passados na linha de comando como uma lista de strings. Assim, podemos verificar qual opção foi passada e chamar a função correspondente.

```
module Main where

import Markers
import System.Environment (getArgs)
import System.FilePath (replaceExtension)
import System.Directory (listDirectory)
import System.FilePath (takeExtension, (</>))

-- Adicionado para diminuir o tamanho do código
wf = writeFile
```

```
raw = convertToRaw
html = convertToHtml
markdown = convertToMarkdown
abnt = convertToAbnt
rpe = replaceExtension

main :: IO ()
main = do
    args <- getArgs
    case args of
    ["-raw", filePath] -> do
        content <- readFile filePath
        wf (rpe filePath ".html") (raw content)
    ["-html", filePath] -> do
        content <- readFile filePath
        wf (rpe filePath ".html") (html content)
    ["-markdown", filePath] -> do
        content <- readFile filePath
        wf (rpe filePath ".md") (markdown content)
    ["-md", filePath] -> do
        content <- readFile filePath
        wf (rpe filePath ".md") (markdown content)
    ["-abnt", filePath] -> do
        content <- readFile filePath
        wf (rpe filePath ".html") (abnt content)
    ["-help"] -> printHelp
    ["-h"] -> printHelp
    _ -> printHelp
```

Note o uso das funções que definimos no The Markers Library, tal como `convertToRaw`. Estas funções serão responsáveis por converter o arquivo Markers para o formato desejado. A função `replaceExtension` é utilizada para trocar a extensão do arquivo, e a função `writeFile` é utilizada para escrever o conteúdo no arquivo.

Com isso definido, podemos compilar o aplicativo utilizando o `stack build`. Esta função irá compilar o projeto e gerar um executável na pasta `src` (caso seja definido como tal, pode ocorrer do arquivo ser gerado na pasta `.stack-work`). Podemos rodar o arquivo acessando a pasta `src`, renomeando o arquivo para `mks` (ou, `mks.exe` caso esteja no Windows) e executando o arquivo gerado com o comando `./mks`. O resultado deve ser o seguinte:

FIGURA 13 – Tela do The Markers Parser.

Fonte: Autor, 2025

Podemos testar o aplicativo criando um arquivo `teste.mks`. Utilizaremos o método de exportação `-raw` para este exemplo. O nosso arquivo terá o seguinte conteúdo:

```
(title)Olá, mundo!(/title)
Este arquivo do Markers irá se transformar em um (b)HTML(/b)!
```

Ao executarmos o comando `mks –raw teste.mks`, será gerado um arquivo na mesma pasta que o comando fora executado, com o nome `teste.html`. O conteúdo do arquivo gerado será o seguinte:

```
<h1>Olá, mundo!</h1>
Este arquivo do Markers irá se transformar em um <b>HTML</b>!
```

Para podermos executar nosso aplicativo por todo o contexto de nossa máquina, devemos adicionar o diretório `src` ao nosso PATH. Note que este processo difere de acordo com o sistema operacional utilizado. No Windows, devemos acessar as configurações de variáveis de ambiente e adicionar o diretório `src` ao PATH. No Linux, podemos adicionar a seguinte linha no arquivo `.bashrc`:

```
export PATH=$PATH:/caminho/para/o/diretorio/src
```

Após isso, devemos reiniciar o terminal para que as alterações tenham efeito. Assim, podemos executar o aplicativo de qualquer lugar do sistema operacional, utilizando o comando `mks` seguido dos parâmetros desejados. Isto nos permitirá utilizar o aplicativo de forma mais prática, sem precisar acessar o diretório `src` toda vez que quisermos executar o aplicativo.

Por fim, também enviaremos este projeto para um repositório no Github, chamado de `parser`, onde será protegido pela licensa do MIT, e irá disponibilizar os arquivos binários prontos para download e uso na máquina do usuário. O repositório do The Markers Parser está disponível em https://github.com/TheMarkersFoundation/parser (Acesso em: 13 maio 2025).

2.4 DESENVOLVIMENTO DO APLICATIVO WEB

Será desenvolvido uma aplicação web que irá explicar as origens do Markers, fornecerá sua documentação, fornecerá um editor em tempo real para o usuário e também irá possuir chamadas API para que qualquer desenvolvedor possa utilizar o Markers em seus projetos. A aplicação web será desenvolvida utilizando o framework `Yesod`, que é um framework web para Haskell. O Yesod é um framework criado por Michael Snoyman que possui uma arquitetura similar à MVC (Model View Controller), e é baseado em mônadas. Será utilizado Haskell, Hamlet (HTML), Lucius (CSS) e Julius (JavaScript) para o desenvolvimento da aplicação.

A aplicação utilizará das bibliotecas `yesod-static` (para manipulação de arquivos estáticos), `file-embed` (para embutir arquivos estáticos no código), `text` e `conduit` (streaming de arquivos), além do próprio The Markers Library.

O website está disponível em https://markers.mirvox.xyz/ (Acesso em: 13 maio 2025).

2.4.1 PÁGINA PRINCIPAL DO SITE

A página principal do site será definida na rota `/` em `routes.yesodroutes`, que irá redirecionar para o `GET Request HomeR`. Este request será definido em `getHomeR` no módulo `Home.hs`. Ele irá conter os widgets de CSS (em Lucius), HTML (em Hamlet) e JavaScript (em Julius). O widget de CSS será responsável por estilizar a página, o widget de HTML será responsável por gerar o conteúdo da página, e o widget de JavaScript será responsável por adicionar interatividade à página.

Usaremos o `yesod-static` para servir os arquivos estáticos, como imagens. Definimos o diretório `static` como o diretório onde os arquivos estáticos serão armazenados. O diretório `static` será criado na raiz do projeto. Será então adicionado em `app/Main.hs` a seguinte linha:

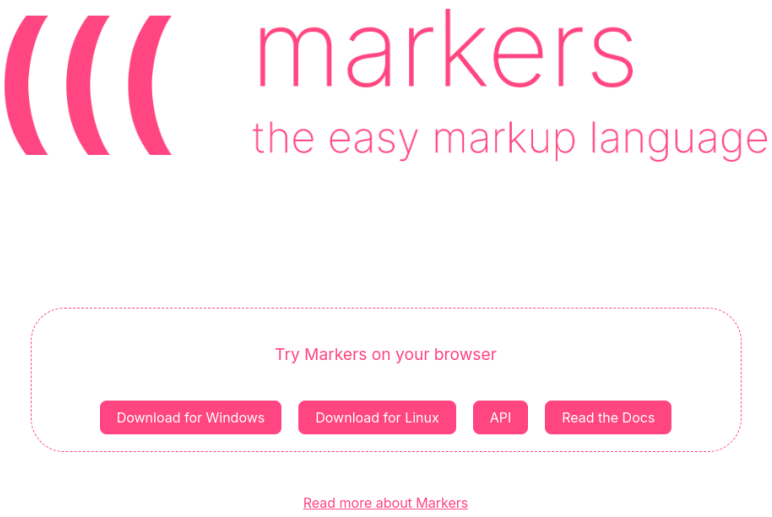
```
main :: IO ()
main = do
staticSite@(Static settings) <- static "static"
warp 5666 App { getStatic = staticSite }
```

Por fim, definiremos a rota de entrega estática em `routes.yesodroutes`, adicionando a linha `/static StaticR Static getStatic`. Podemos então chamar os arquivos estáticos de nossa pasta `static` pelo Hamlet da seguinte maneira:

```
-- ...
[whamlet|
    <div .header>
        <img width="13%" src=@{StaticR logo3_png}>
        <img width="40%" src=@{StaticR logo2_png}>
        <!-- ... -->
```

O resultado final da página principal ficará da seguinte maneira:

FIGURA 14 – Página principal da Web do Markers.



Fonte: Autor, 2025

Podemos inserir arquivos do Markers diretamente em nossa página e renderizá-los nativamente. Para isso, adicionaremos o TheMarkersLibrary no topo de nosso arquivo e criaremos uma função chamada `getIntroducingMarkers`.

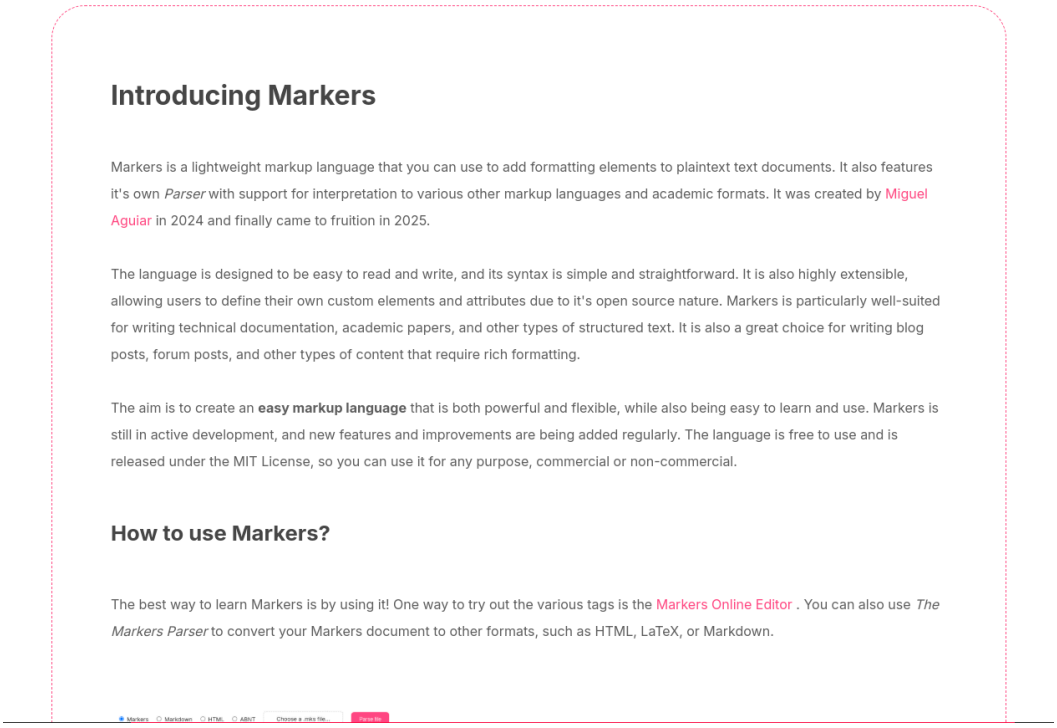
```
import Foundation
import Yesod.Core
import Yesod.Static
import Data.FileEmbed (embedStringFile)
import Markers

getIntroducingMarkers :: String
getIntroducingMarkers = convertToRaw
                        $(embedStringFile
                        "static/Introducing_Markers.mks")
```

Esta função irá ler o arquivo `Introducing_Markers.mks` e convertê-lo para HTML sem estilo. Podemos então utilizar esta função em nosso widget do Hamlet, inclusive estilizando-o como quisermos com ajuda das classes CSS definidas no momento da interpretação no capítulo 2.2.4. Observe como podemos renderizar o arquivo Markers diretamente em nossa página web:

```
<div .markersContainer id="introducingMarkers">
    ^{preEscapedToMarkup getIntroducingMarkers}
```

Aqui, utilizamos a função `preEscapedToMarkup` para evitar que o HTML gerado seja escapado, permitindo que o HTML seja renderizado corretamente na página. Podemos então estilizar o conteúdo da página como quisermos, utilizando CSS e JavaScript. O resultado final da página ficará da seguinte maneira:

FIGURA 15 – Arquivo Markers renderizado.



Fonte: Autor, 2025

Com isto, demonstramos as habilidades do Markers em renderizar o arquivo diretamente na página web, sem a necessidade de conversão para HTML.

2.4.2 DESENVOLVIMENTO DAS APIS

A aplicação web também irá fornecer APIs para que qualquer desenvolvedor possa utilizar o Markers em seus projetos. As APIs serão desenvolvidas utilizando o Yesod, e serão baseadas em requisições REST. As APIs irão permitir que os desenvolvedores possam enviar arquivos Markers e receber o arquivo convertido para o formato desejado. As APIs também usarão para realizar o parsing e a conversão da chamada.

As chamadas APIs receberão um request do tipo `POST` cujo cabeçalho seja um texto do tipo `text/raw`. O texto fornecido deve ser um documento Markers, e o retorno será o arquivo convertido para o formato desejado. O retorno será do tipo `text/html`.

A API será definida no arquivo `src/Parsing.hs`, onde as rotas relacionadas ao *parsing* das requisições estarão disponíveis. Nossa função será definida da seguinte forma:

```
postConvertMksHtmlR :: Handler Html
postConvertMksHtmlR = do
    lbs <- runConduit $ rawRequestBody .| sinkLbs
    let raw = TL.toStrict (TLE.decodeUtf8 lbs)
    let normalizedRaw = T.replace "\r\n" "\n" raw
    let html = preEscapedToMarkup (convertToHtml
               (T.unpack normalizedRaw))
    sendResponse (toTypedContent html)
```

Esta função recebe um request do tipo `POST`, lê o corpo do request como uma string, normaliza as quebras de linha, converte o arquivo Markers para HTML utilizando a função `convertToHtml`, e retorna o resultado como uma resposta HTML, cujo conteúdo é o arquivo interpretado para o HTML estilizado.

Para que a função funcione corretamente, devemos adicionar as seguintes importações no topo do arquivo:

```
import Foundation
import Yesod.Core
import Markers

import Conduit (runConduit, (.|))
import Data.Conduit.Binary (sinkLbs)
import qualified Data.Text.Lazy.Encoding as TLE
import qualified Data.Text.Lazy as TL
import qualified Data.Text as T
```

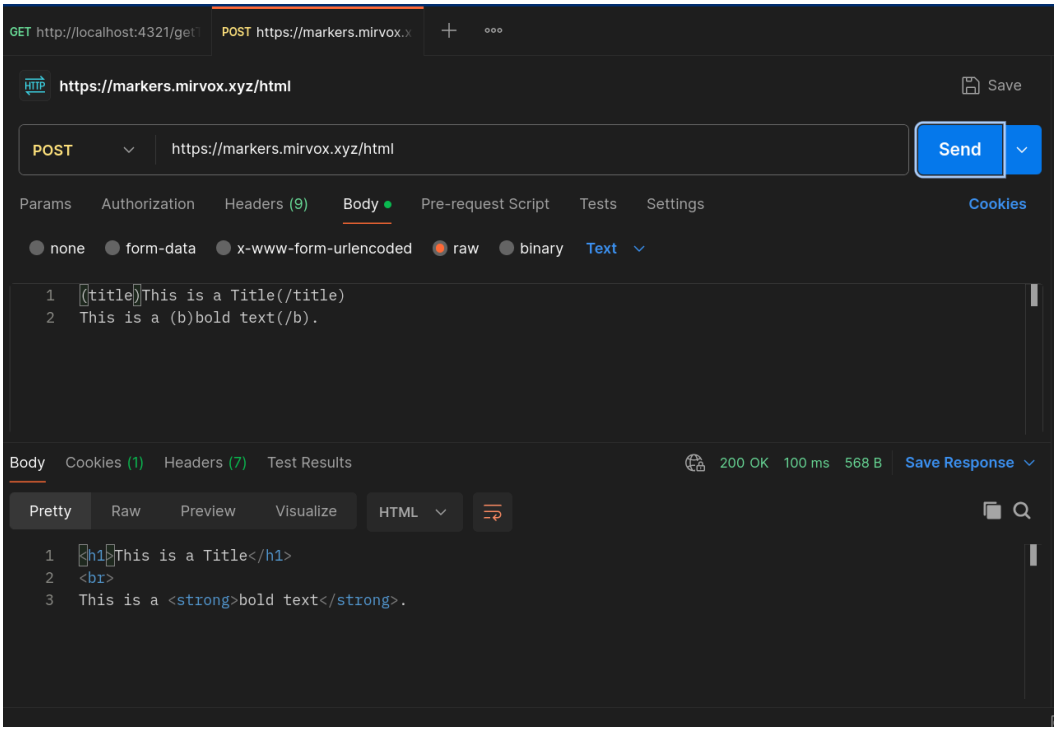
O mesmo processo é repetido para os outros métodos de interpretação, mudando o nome da função e a chamada responsável pelo seu respectivo tipo.

Será então adicionado ao arquivo `routes.yesodroutes` as rotas necessárias para a chamada da API, sendo estes:

```
/markers           ConvertMksHtmlR POST
/md                ConvertMksMarkdownR POST
/html              ConvertMksRawR POST
/abnt              ConvertMksAbntR POST
```

Podemos testar nossa API realizando uma requisição para um destes *endpoints* utilizando uma ferramenta como o Postman. O Postman é uma plataforma completa para desenvolvimento, teste e documentação de APIs. Podemos utilizá-la para fazer nossa requisição e *debugar* caso necessário.

FIGURA 16 – Requisição do Postman no endpoint /html.



Fonte: Autor, 2025

2.4.3 EDITOR DE TEXTO ONLINE

Será desenvolvido um editor de texto online que permita com que o usuário da aplicação veja a conversão em tempo real de seu arquivo Markers. O editor de texto será feito com a adição do Julius para adicionar JavaScript ao HTML do Yesod. Criaremos uma nova rota na aplicação web, chamada de `getParserR` .

FIGURA 17 – Editor de texto online.

Fonte: Autor, 2025

Como observamos na **Figura 17**, nesta página utilizaremos de um elemento `textarea` do HTML para que o usuário possa digitar seu arquivo Markers. Também adicionaremos um botão de envio de arquivos, para que o usuário possua a habilidade de enviar um arquivo Markers já existente e vê-lo pelo navegador. Além disso, no topo da página, podemos observar a inclusão de quatro botões `radio` representantes dos modelos de *parseamento* do arquivo, sendo estes: `HTML`, `Markdown`, `Raw` e `ABNT`. Estes botões serão utilizados para definir o tipo de conversão que será realizada no arquivo Markers. Ao lado direito do `textarea`, observamos a inclusão de uma área de visualização do arquivo convertido, que será atualizada em tempo real conforme o usuário digita ou envia um arquivo Markers.

A conversão do documento será feita utilizando da função `addEventListener` do JavaScript. Utilizaremos esta função para "escutar" alterações no `DOM` (Document Object Model) do HTML, e assim, realizar a conversão do arquivo Markers para o formato desejado. A função `addEventListener` será utilizada para "escutar" o evento de `change` dos botões de conversão, além da entrada de arquivos. Após qualquer uma destas mudanças ocorrer, iremos realizar a conversão do arquivo Markers para o formato desejado, utilizando a API que criamos anteriormente. A função `fetch` do JavaScript será utilizada para realizar a requisição para a API, passando o arquivo Markers como corpo da requisição.

Será assim então populado a área de visualização com o resultado da conversão, utilizando a função `innerHTML` do JavaScript.

2.4.4 HOSPEDANDO A APLICAÇÃO

Hospedaremos a aplicação web utilizando o Docker. O Docker é uma plataforma de software que permite criar, implantar e executar aplicativos em contêineres. Os contêineres são pacotes leves e portáteis que incluem tudo o que um aplicativo precisa para ser executado, incluindo o código, as bibliotecas e as dependências.

O container Docker será construído utilizando o Dockerfile, que é um arquivo de texto que contém todas as instruções necessárias para criar o container. O Dockerfile será utilizado para instalar todas as dependências necessárias para a aplicação web, e também para copiar os arquivos da aplicação para dentro do container.

Nosso Dockerfile será enviado para uma VPS (Virtual Private Server). A VPS é um servidor virtual que simula um servidor dedicado, permitindo que possamos hospedar nossa aplicação web de forma segura e escalável. A VPS será utilizada para hospedar o container Docker, permitindo que possamos escalar nossa aplicação de acordo com a demanda.

2.5 MIRAGEM: O EDITOR DO MARKERS

Será desenvolvido um editor de texto com funcionalidades de conexão com a internet e armazenamento de documentos Markers denominado Miragem. Este aplicativo será de código fechado (*closed source*). Será utilizado JavaScript, HTML e CSS. Para banco de dados, será utilizado o banco NoSQL MongoDB. Usaremos as bibliotecas `mongo` para conexão com o banco de dados, `bcrypt` para criptografia de senhas e `dotenv` para variáveis de ambiente.

O objetivo da aplicação é simplificar a criação de documentos Markers, permitindo que o usuário possa criar, editar e salvar documentos Markers de forma simples e rápida, garantindo uma experiência fluida tanto para iniciantes quanto para usuários avançados. Através de uma interface intuitiva, o Miragem busca reduzir ao máximo a curva de aprendizado, oferecendo uma pré-visualização do documento, e templates configuráveis conforme as normas ABNT e outras formatações desejadas.

A aplicação também irá suportar o envio dos arquivos para o banco de dados, permitindo que o usuário possa acessar seus documentos de qualquer lugar, além de compartilha-los com outros usuários via uma URL. Também implementaremos o conceito de *Ruina*.

Uma Ruina é uma coleção de documentos Markers que possuam o mesmo contexto. Podemos equiparar as Ruinas aos blocos do aplicativo Notion, onde o usuário pode criar uma coleção de documentos que possuam o mesmo contexto. O usuário poderá criar Ruinas e adicionar documentos a elas, permitindo que o usuário possa organizar seus documentos de forma simples e rápida. Em essência, são como uma pasta de arquivos que podem ser compartilhadas com outros usuários. O usuário poderá criar Ruinas públicas ou privadas, permitindo que o usuário possa compartilhar seus documentos com outros usuários de forma simples e rápida.

A aplicação será desenvolvida utilizando o framework Electron, que possibilita a criação de aplicações desktop multiplataforma com tecnologias web. Com o Electron, podemos empacotar toda a lógica do Miragem em um executável leve, mantendo a aparência nativa em Windows, macOS e Linux. Apesar de suportar bibliotecas como React ou Vue.js, optou-se por não utilizá-las para manter o bundle enxuto e reduzir dependências, priorizando performance e menor consumo de memória.

A lógica da aplicação foi organizada em três arquivos principais:

1. main.js: Inicializa o Electron, gerencia janelas e faz a conexão com o MongoDB. Também responde a eventos do sistema (por exemplo, salvar ao fechar a janela).

2. renderer.js: Responsável pela renderização da interface e pela comunicação com o backend via `ipcRenderer`. Trata eventos de usuário (cliques, atalhos de teclado) e dispara chamadas locais, como parsing do documento Markers para preview e exportação em PDF/HTML.

3. preload.js: Atua como ponte segura entre o contexto principal (Node.js) e o front-end, expondo apenas APIs restritas (por exemplo, funções de leitura/gravação de arquivos e acesso ao banco) para evitar vulnerabilidades de segurança.

FIGURA 18 – Editor Miragem.

Fonte: Autor, 2025

Ao abrir a aplicação, o usuário é recebido por uma tela de boas-vindas com o logotipo do Miragem e um módulo de dicas rápidas que pode ser desativado. A interface de boas-vindas possui:

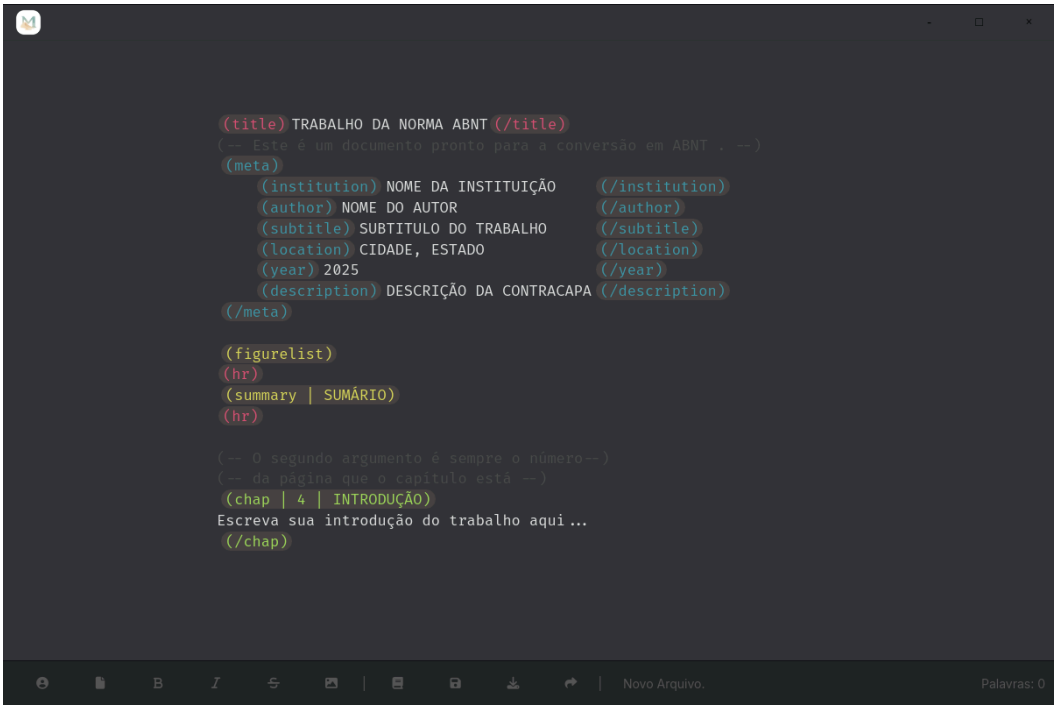
- Botão “Começar” para criar um novo documento ou abrir um existente

- Lista de documentos recentes, exibindo título, data de modificação e contagem de palavras

- Opção de tutorial interativo para primeiros passos

Após clicar em “Começar”, o usuário acessa a tela de edição, que apresenta um editor de texto; baseado em `<textarea>`, enriquecido com a biblioteca CodeMirror, oferecendo destaque de sintaxe, indentação automática e suporte a plugins de autocomplete.

FIGURA 19 – Editor Miragem.



Fonte: Autor, 2025

Na **Figura 19** observamos a interface principal do editor da aplicação. Na parte inferior da tela, há uma barra de status com botões de ação rápida; sendo estes: Configurações da Conta, Novo Arquivo, Negrito, Itálico, Tachado, Adicionar Imagem, Abrir a Biblioteca, Salvar Arquivo, Carregar Arquivo e Parsear. Além disso, do lado oposto dos botões de acesso rápido, temos um contador de palavras em tempo real.

Podemos notar que o editor de texto já se encontra preenchido com um arquivo Markers de exemplo. Este exemplo foi construido cuidadosamente junto com as diferentes cores das tags para dar um sentimento de familiaridade com o usuário. O exemplo em sí é um arquivo markers válido que já serve como base para a construção de um documento acadêmico da norma ABNT. Ao clicarmos nos botões de ação rápida, podemos observar que o editor de texto se comporta como esperado, permitindo que o usuário possa formatar o texto de acordo com suas necessidades. Por exemplo, clicar no botão de Negrito irá adicionar a tag `(b)(/b)` ao texto selecionado.

O botão de "Adicionar Imagem" permite que o usuário adicione uma imagem ao documento. Este botão abrirá uma caixa de diálogo para que o usuário possa selecionar a imagem que deseja adicionar, além de escrever sua descrição. A imagem será adicionada ao

documento como uma tag `(img)`, seu atributo sendo a imagem enviada convertida para base64. A caixa de diálogo pode ser observada na **Figura 20** a seguir.

FIGURA 20 – Diálogo de adição de imagem.

Fonte: Autor, 2025

Clicando no botão de salvar, o usuário será questionado onde deseja salvar o arquivo, abrindo a janela do sistema operacional para seleção de arquivos. O arquivo será salvo com a extensão `.mks` e conterá o conteúdo do editor de texto. O usuário também poderá carregar um arquivo já existente, clicando no botão "Carregar Arquivo". Isto abrirá uma janela do sistema operacional para que o usuário possa selecionar o arquivo Markers que deseja carregar. O conteúdo do arquivo será carregado no editor de texto, permitindo que o usuário possa editar o documento.

Caso o documento já esteja salvo no disco, e aberto no editor, clicar no botão "Salvar Arquivo" irá salvar o documento no mesmo local onde ele foi aberto, sobrescrevendo o arquivo existente, sem a necessidade de abrir a janela de seleção de arquivos novamente. Isto permite que o usuário possa salvar o documento rapidamente, sem precisar passar pelo processo de seleção de arquivos toda vez que quiser salvar. Os atalhos de teclado também foram implementados para facilitar a experiência do usuário, permitindo que o usuário possa salvar o documento utilizando `Ctrl + S` ou `Cmd + S` no MacOS.

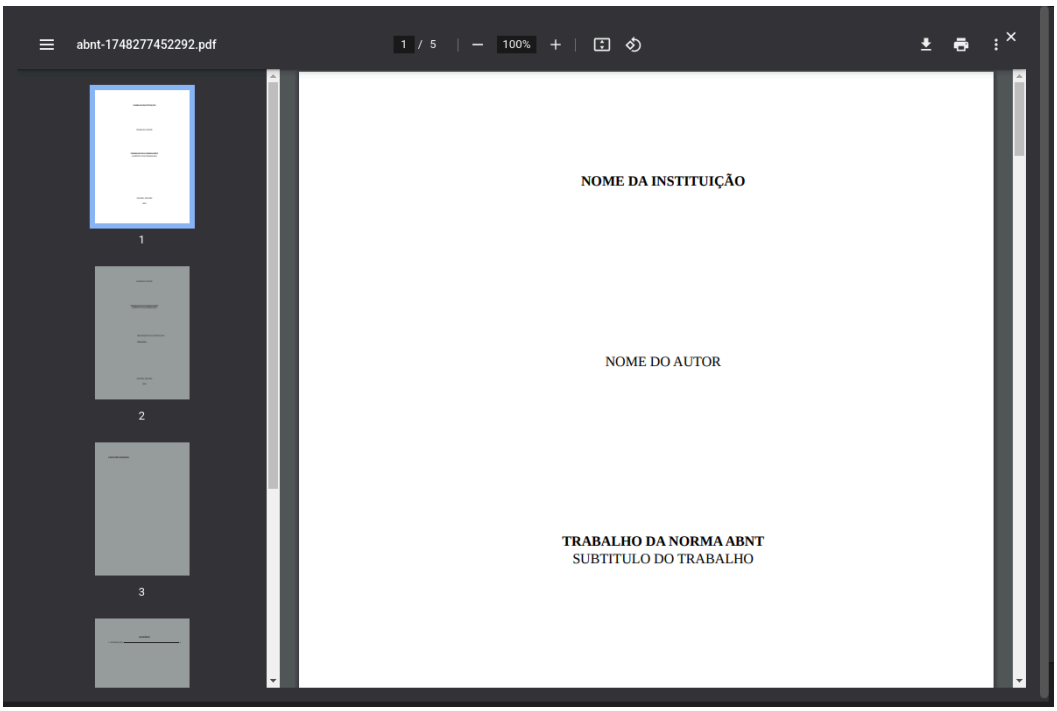
Ao clicar no ícone da seta (Parsear) - será aberta a função para exportar o documento, possuindo duas seleções, "ABNT" e "Markers". O mesmo é uma caixa de diálogo similar a do Google Chrome, como demonstrado na **Figura 21**.

FIGURA 21 – Seleção de Exportação.

Fonte: Autor, 2025

A conversão do arquivo ocorre no momento que o usuário clica no botão de exportar. No momento que isso é feito, uma requisição API é feita para o endpoint `markers.mirvox.xyz/abnt` ; esta requisição, como visto no capítulo 2.4.2, irá receber um `text/raw` - que, no caso, será o texto que o usuário digitou no editor, e irá retornar um HTML formatado. Isto então é mostrado para o usuário, caso a opção selecionada seja "Markers", será mostrado o HTML gerado para o usuário, caso a seleção seja "ABNT", será mostrado o arquivo PDF que fora convertido, como observado na **Figura 22**.

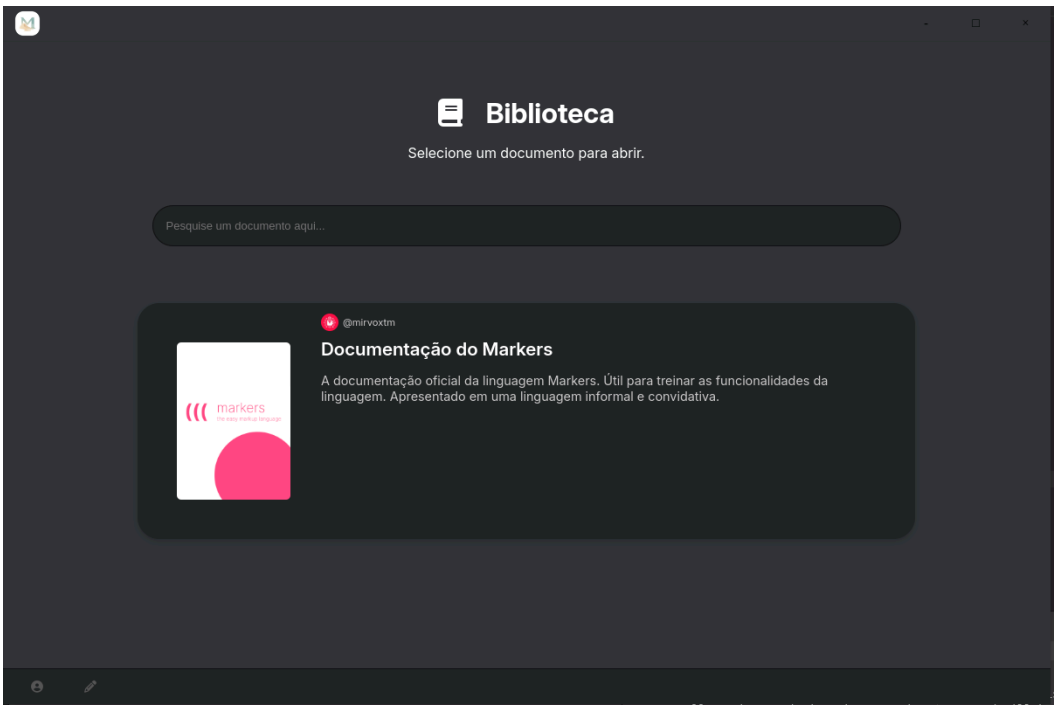
FIGURA 22 – Arquivo ABNT no Miragem.



Fonte: Autor, 2025

O botão "Abrir Biblioteca" redireciona o usuário para uma página que possui todos os documentos publicados na plataforma, permitindo que o usuário possa acessar documentos de outros usuários que foram disponibilizados para leitura sob-demanda. Os documentos publicados são mostrados em uma lista, que possui o titulo do documento, o autor, sua capa e descrição. Além disso, no rodapé possuimos dois botões, sendo estes "Configurações da Conta" e "Abrir Editor". Observamos isto na **Figura 23**.

FIGURA 23 – Biblioteca do Miragem.



Fonte: Autor, 2025

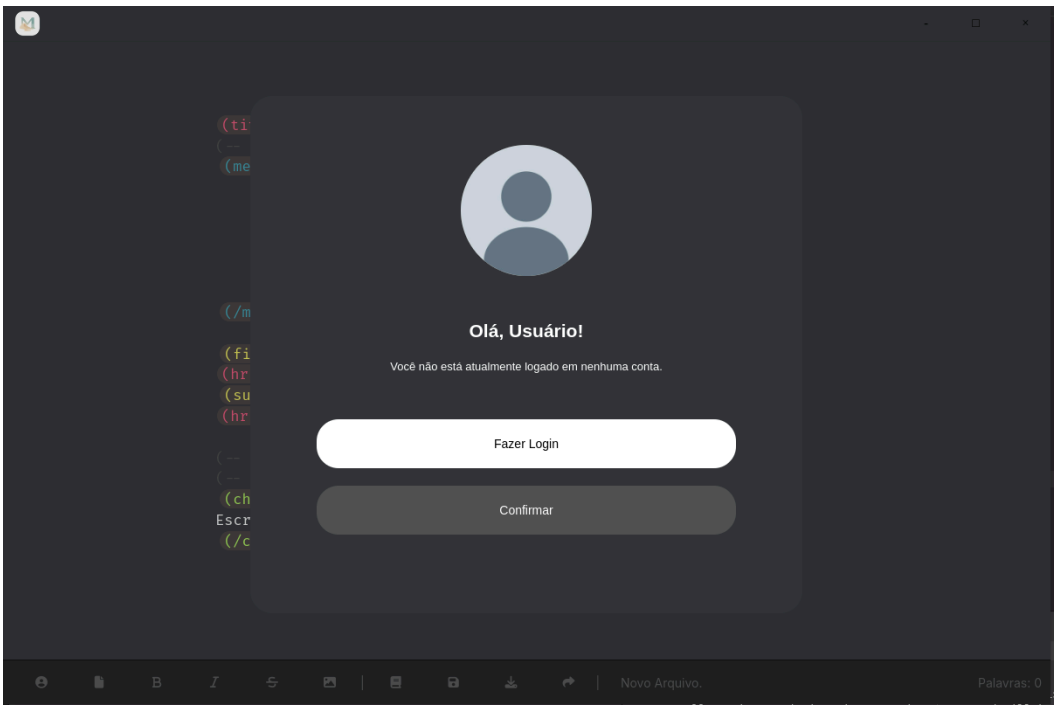
Ao clicar no elemento presente na lista, será aberto sua visualização completa e interpretada com base no tipo que o documento fora publicado. Isto pode ser visto na **Figura 24** a seguir.

FIGURA 24 – Biblioteca do Miragem.

Fonte: Autor, 2025

Ao clicar em "Configurações da Conta", o usuário observará uma caixa de diálogo informando-o que o mesmo não está logado, e que para acessar as configurações da conta, o usuário deve se registrar ou fazer login. Nesta tela observamos uma imagem padrão de usuário, uma mensagem informando o usuário que o mesmo não possui acesso a sua conta atualmente, e dois botões; um para fazer login, e outro para confirmar. Visualizamos isto na **Figura 25**.

FIGURA 25 – Configurações da Conta.



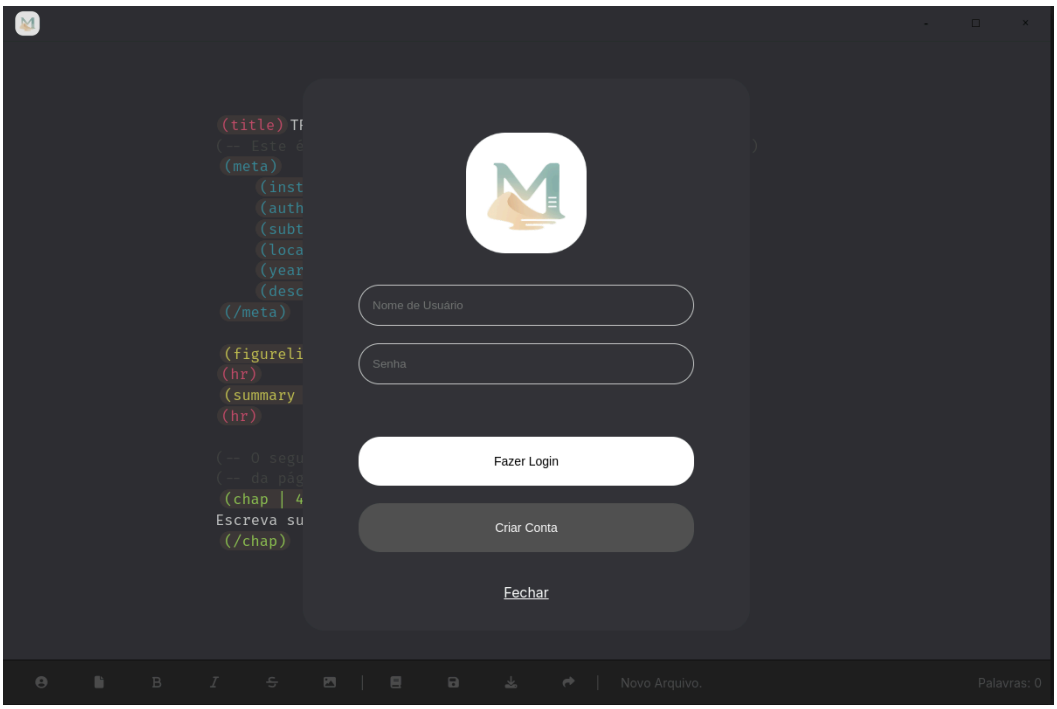
Fonte: Autor, 2025

Clicando em "Confirmar", o diálogo será fechado e o usuário retornará para a tela que estava anteriormente. Ao clicar no botão "Fazer Login", será aberto outro diálogo que irá pedir as informações de acesso do usuário, sendo estes o seu *handle* e sua senha. O usuário também poderá se registrar clicando no botão "Criar Conta".

O diálogo de login é mostrado na **Figura 26**. Caso o usuário falhe em inserir as credenciais corretas, o erro presente da **Figura 27** será mostrado. Ao clicar em "Criar Conta", o usuário será redirecionado para uma página de registro, onde poderá criar sua conta.

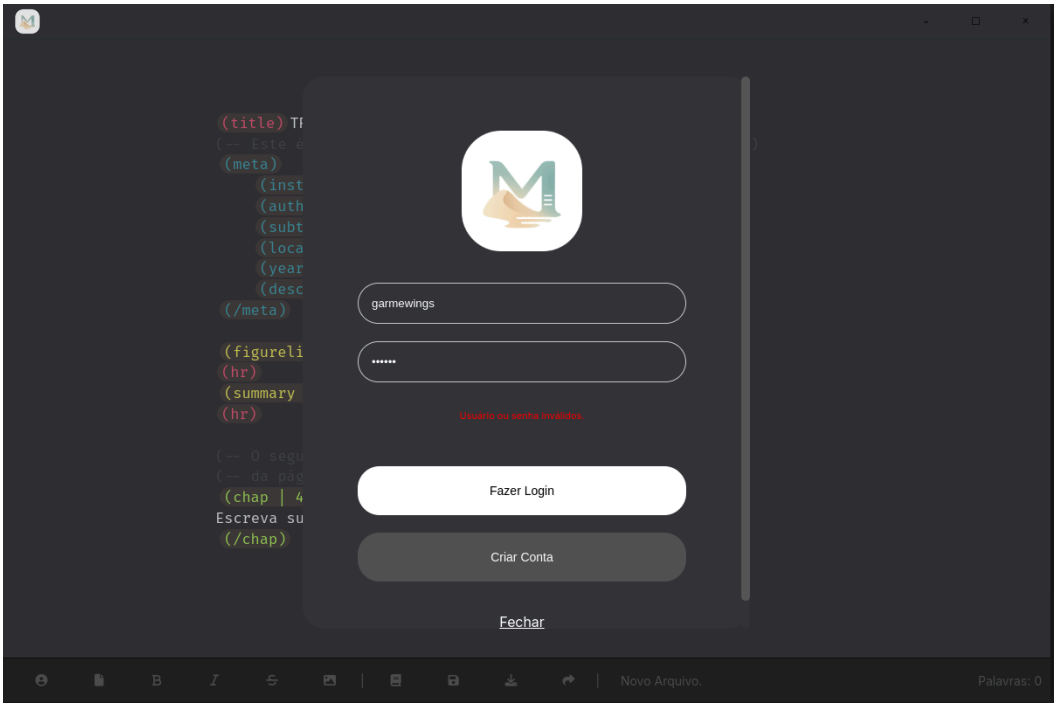
O registro é feito através do envio de um formulário com as informações do usuário. Ao clicar na imagem de exemplo, o usuário poderá adicionar uma imagem de perfil, que será salva no banco de dados (sendo convertida para base64) e associada ao usuário. Outros campos necessários são o nome de usuário (que será convertido para um *handle*), seu E-Mail e senha. A página de registro é mostrada na **Figura 28**.

FIGURA 26 – Tela de Login.



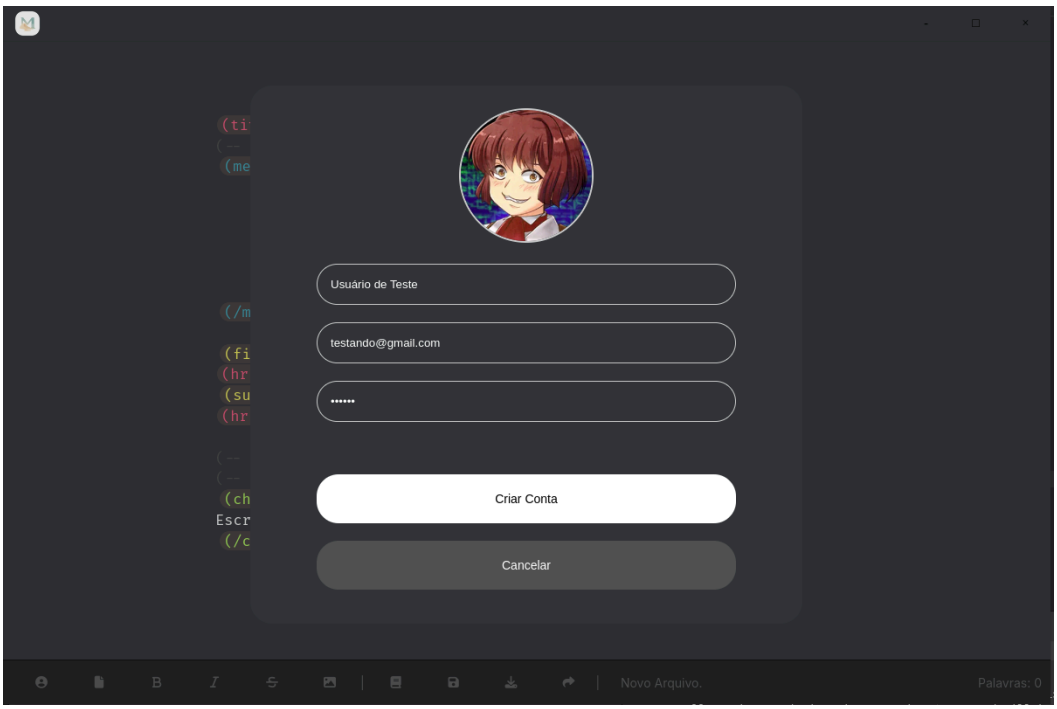
Fonte: Autor, 2025

FIGURA 27 – Erro de Login.



Fonte: Autor, 2025

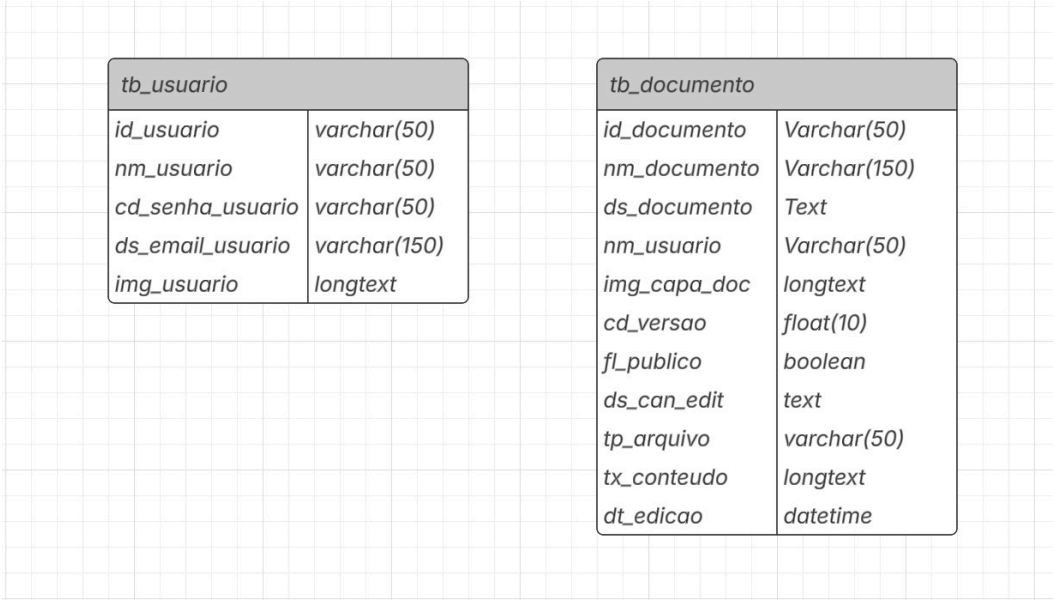
FIGURA 28 – Tela de Registro.



Fonte: Autor, 2025

Caso o usuário exista, e consiga efetuar o login com sucesso, a aplicação será atualizada, agora mostrando a foto de perfil do usuário no botão de Configurações da Conta. Um novo botão será adicionado ao rodapé do editor, sendo este "Salvar na Nuvem". Este botão irá salvar o documento do usuário no banco de dados, permitindo que o usuário possa acessar o documento de qualquer lugar. O documento será automaticamente salvo com o título dado pelo usuário entre as tags `(title)`. Podemos observar que a modelagem de nosso banco de dados é simples, similar ao que pode ser observado na **Figura 29**. Note que esta representação é simplificada, dado que a estrutura real do banco de dados em MongoDB é baseada em documentos - sendo estes, arquivos JSON, e não em tabelas como em bancos relacionais. Além do fato que representar um banco de dados atualmente em uso seria inseguro e antiético, visto que poderia expor dados sensíveis de usuários.

FIGURA 29 – Exemplo do modelo de Banco de Dados.

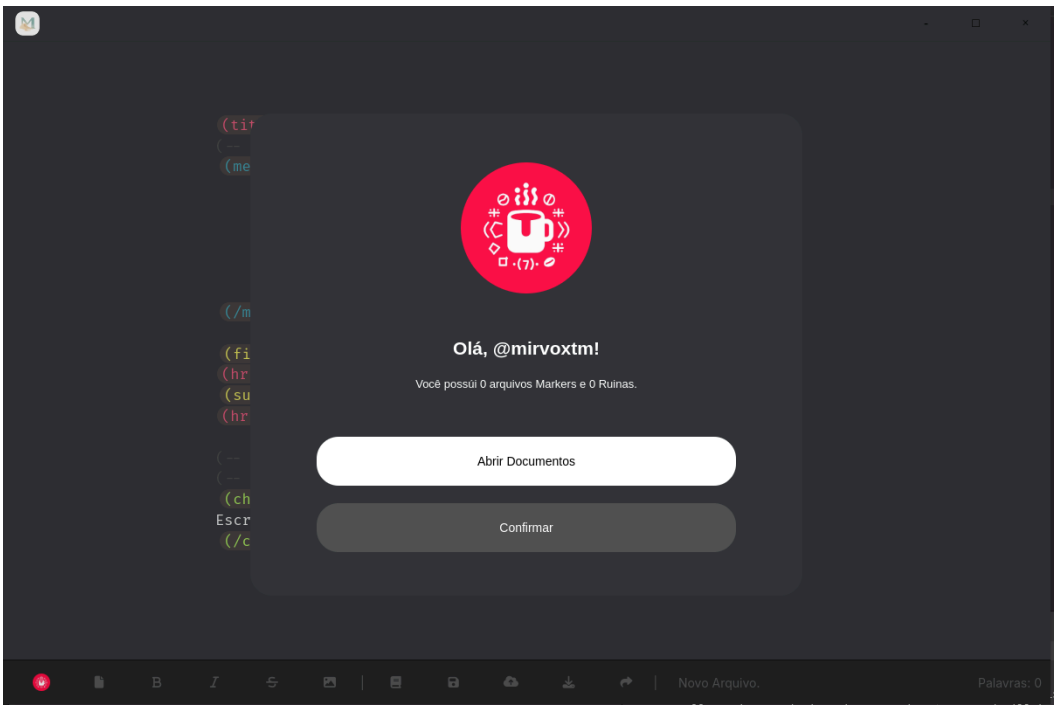


Fonte: Autor, 2025

Observamos que um documento, ao ser salvo na nuvem, receberá um ID único, que será utilizado para identificar o documento no banco de dados. O campo `nm_documento` será populado com o título do documento, e o campo `nm_autor` será populado com o handle (nome de usuário). O campo `tx_conteudo` é populado com o conteúdo do documento, que será o texto Markers. O resto dos campos serão, por hora, ignorados, já que não são necessários para o salvamento do documento.

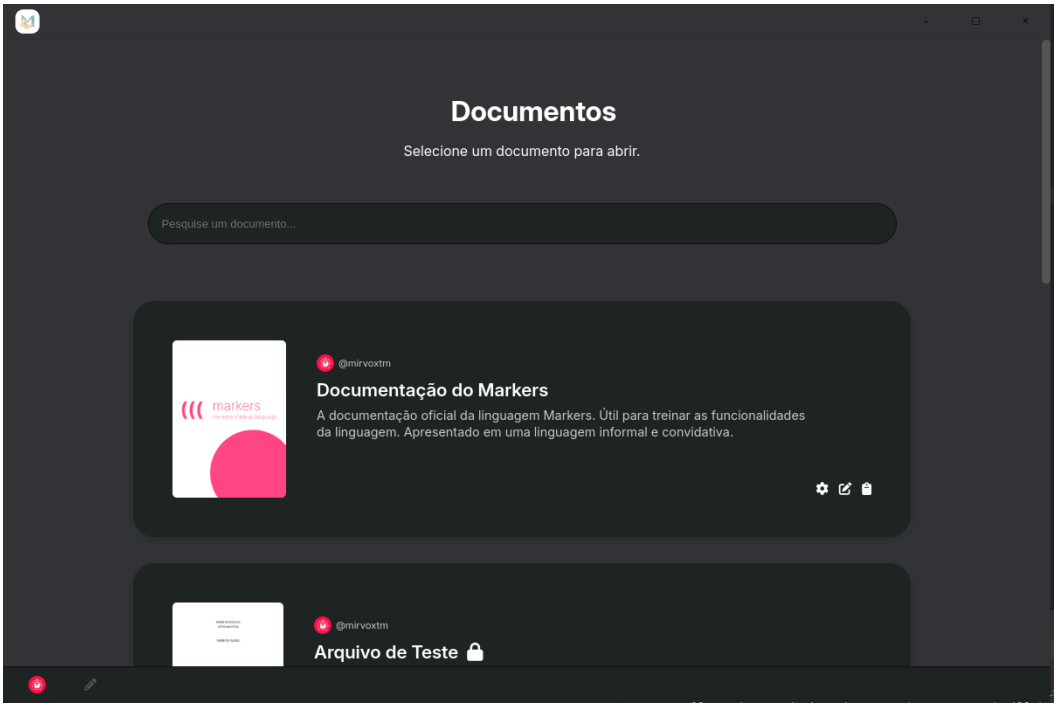
Os documentos salvos na nuvem podem ser acessados através do botão "Abrir Documentos" no diálogo de Configurações da Conta, como pode ser observado na **Figura 30**. Aqui serão listados todos os documentos salvos pelo usuário (ou seja, sejam associados ao `nm_usuario` (handle) do usuário atualmente logado). O usuário poderá abrir o documento clicando no cartão. Dentro do mesmo, podem ser vistos três ícones, sendo estes "Configurar Arquivo", "Editar Online" e "Copiar Link".

FIGURA 30 – Tela de Usuário Logado.



Fonte: Autor, 2025

FIGURA 31 – Tela de Documentos.

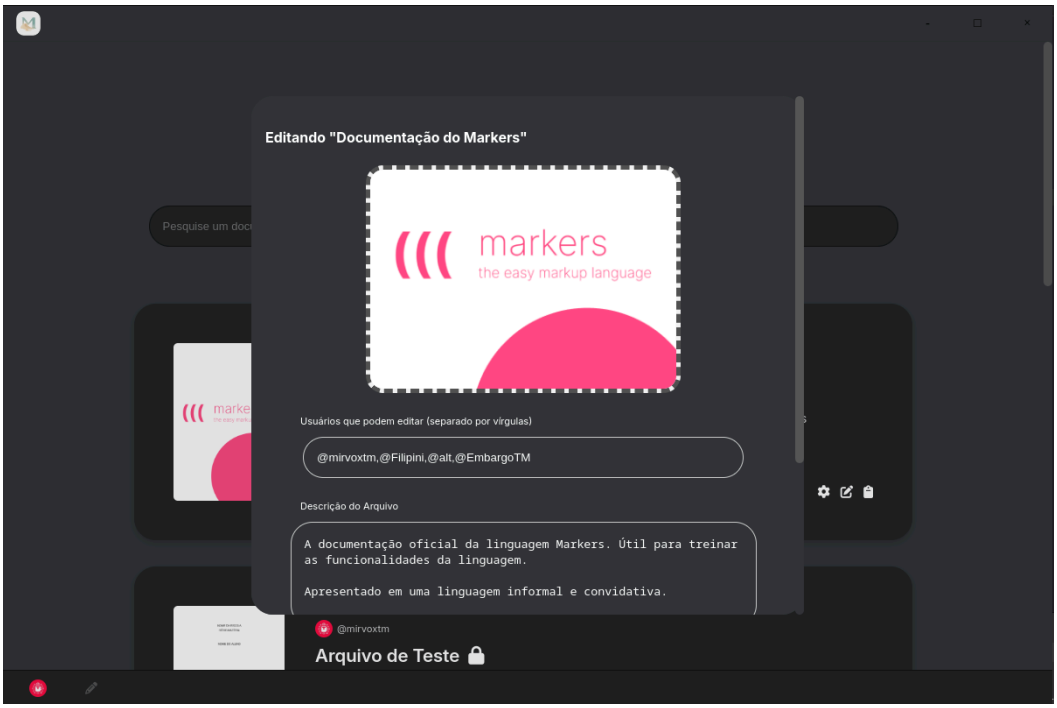


Fonte: Autor, 2025

O botão "Configurar Arquivo" abre uma caixa de diálogo com opções extras para o arquivo atualmente salvo. Este permite a adição de descrição, capa, e a possibilidade de tornar

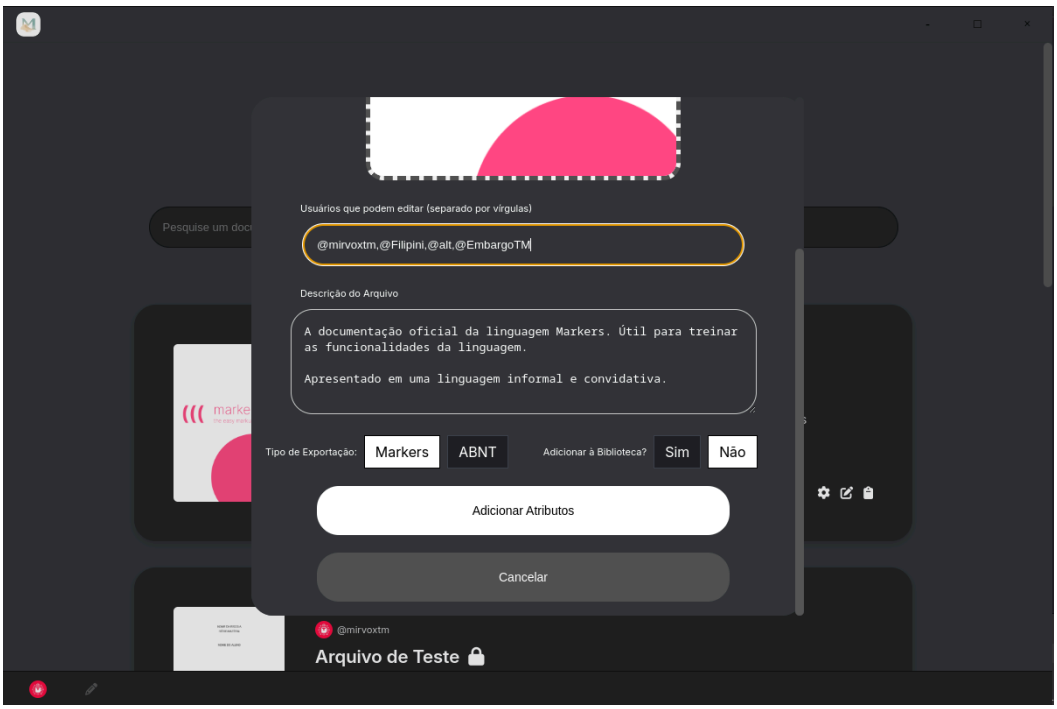
o arquivo público ou privado. Além disso, ele também permite que o usuário adicione outros usuários do Miragem que podem editar este arquivo, isto irá mostrar o documento para os usuários selecionados, e permitir que os mesmos possam editá-los online. Podemos observar este diálogo na **Figura 32 e 33**. Ao clicar em "Adicionar Atributos", os dados serão salvos no banco de dados, e o usuário será redirecionado para a tela de documentos novamente.

FIGURA 32 – Tela de Configuração de Documento.



Fonte: Autor, 2025

FIGURA 33 – Tela de Configuração de Documento.

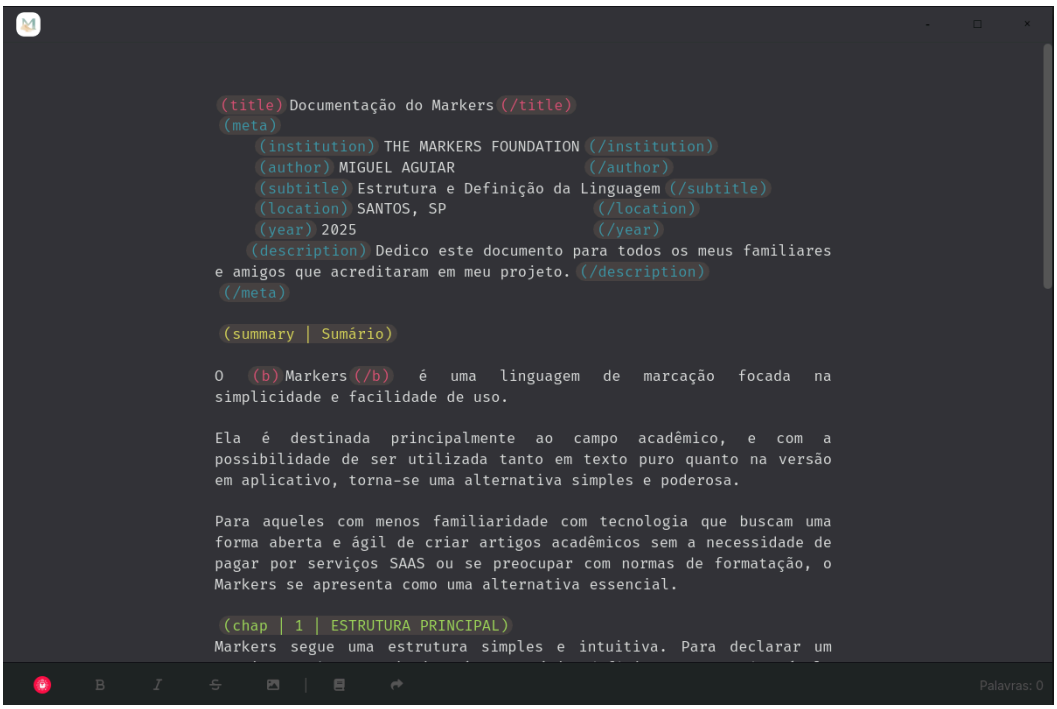


Fonte: Autor, 2025

O botão "Editar Online" redireciona o usuário para o editor online, onde o usuário poderá editar o documento em tempo real. Sua tela pode ser vista na figura 33. Este sistema utiliza da integração de um WebSocket para permitir que o usuário possa editar o documento em tempo real, e ver as alterações feitas por outros usuários. O WebSocket é uma tecnologia que permite a comunicação bidirecional entre o cliente e o servidor. O WebSocket utilizado foi o `yjs`. Para seu funcionamento, foi necessário sua instalação e configuração nos servidores que hospedam e armazenam tudo relacionado a linguagem Markers. O yjs é uma biblioteca que utiliza o conceito de *CRDT* (Conflict-free Replicated Data Type), que permite que múltiplos usuários possam editar o mesmo documento ao mesmo tempo, sem conflitos. Este é utilizado para sincronizar as alterações feitas por cada usuário, permitindo que todos vejam as alterações em tempo real.

Este mesmo modelo de edição online é utilizado por outros editores de texto, como o Google Docs e o Notion. Disponibilizando esta tecnologia para o Miragem, permitimos que os usuários possam colaborar em documentos de forma simples e rápida, sem a necessidade de enviar arquivos por e-mail ou utilizar outras ferramentas de colaboração. A tela do editor online é similar ao editor offline, porém sua funcionalidade no backend é diferente, visto que o editor online utiliza o WebSocket para sincronizar as alterações feitas por cada usuário, sendo assim, salvando o documento no banco de dados a cada atualização.

FIGURA 34 – Editor Online do Miragem.



Fonte: Autor, 2025

O Miragem também conta com uma configuração única de protocolo, sendo este o `markers://` . Este protocolo é utilizado para abrir arquivos Markers diretamente no editor, ou mostrá-lo interpretado diretamente. Assim, permitindo que o usuário possa abrir arquivos Markers diretamente do navegador, ou de outros aplicativos. Este protocolo foi configurado pelo Electron, e podemos copiar o link de um arquivo salvo no Miragem com facilidade clicando no botão "Copiar Link".

O link copiado terá o formato `markers://id/método` . Onde o id é o ID do documento salvo no banco de dados, e o método é o tipo de conversão que será realizada no arquivo. Por exemplo, `markers://id/abnt` irá abrir o documento com a conversão para ABNT, enquanto `markers://id/markers` irá abrir o documento com a conversão para Markers. O usuário poderá abrir este link em qualquer navegador, ou aplicativo que suporte o protocolo `markers://` , e o Miragem irá abrir o documento com a conversão desejada instantâneamente.

Tendo demonstrado suas funcionalidades, será disponibilizado sua versão para download de forma online no site oficial do Markers, hospedado na URL https://markers.mirvox.xyz. A versão para download será disponibilizada para Windows, macOS e Linux, permitindo que qualquer usuário possa utilizar o Miragem em seu sistema operacional preferido (Acesso em 29 maio 2025).

3 ESTUDO DE CASOS DE USO

Foi levantado um caso de uso da linguagem Markers com amigos e familiares que possuem interesse em escrever livros, artigos e outros documentos acadêmicos. Uma aluna do curso de ciência de dados da Faculdade de Tecnologia da Baixada Santista Rubens Lara também utilizou da linguagem para o desenvolvimento de um trabalho de conclusão de curso. Para os participantes, um questionário de sete perguntas foi aplicado para entender como a linguagem Markers era utilizada por estes usuários.

Questão	**Entrevistada 1**
Qual sua experiência com linguagens de marcação?	N/A
O que você acha da linguagem Markers? Considera-a fácil, ou difícil? Qual a sua experiência com a linguagem?	Fácil, simples e prática de usar
Quais tipos de documentos você geralmente escreve com a linguagem?	Trabalhos acadêmicos para faculdade
Qual método você prefere para escrever os documentos?	Aplicação CLI
Qual método de exportação você mais usa?	ABNT
Você acha que o Markers oferece uma experiência mais fácil para o desenvolvimento de artigos acadêmicos e documentos?	Sim, facilita muito
O que você acha que pode melhorar na linguagem e em suas aplicações?	Acrescentar algo para escape de texto, para permitir que o texto não tenha conflito com nenhuma tag

Questão	Entrevistado 2
Qual sua experiência com linguagens de marcação?	Markers é a minha primeira linguagem de marcação, mas já faz tempo que eu uso regularmente programas de edição de texto.
O que você acha da linguagem Markers? Considera-a fácil, ou difícil? Qual a sua experiência com a linguagem?	Considero uma linguagem bem fácil e intuitiva. Apesar de ter pouco tempo com ela, já foi o suficiente para eu ter uma noção completa de como usar
Quais tipos de documentos você geralmente escreve com a linguagem?	Documentos informais e pessoais, mas eventualmente eu estarei utilizando em trabalhos acadêmicos
Qual método você prefere para escrever os documentos?	Parser Online e Miragem, mas principalmente o Miragem
Qual método de exportação você mais usa?	Markers
Você acha que o Markers oferece uma experiência mais fácil para o desenvolvimento de artigos acadêmicos e documentos?	Sim, com certeza. Toda a sua estrutura é baseada nisso
O que você acha que pode melhorar na linguagem e em suas aplicações?	Apenas aplicação de imagens, áudio e vídeo para a linguagem. E formatação de letra e configuração de pagina para as aplicações

Questão	Entrevistado 3
Qual sua experiência com linguagens de marcação?	Não tenho experiência com nenhuma linguagem de marcação.
O que você acha da linguagem Markers? Considera-a fácil, ou difícil? Qual a sua experiência com a linguagem?	Achei bem fácil, depois de entender como funciona.
Quais tipos de documentos você geralmente escreve com a linguagem?	Não tenho utilidade no momento, mas eu utilizaria para escrever artigos ou trabalhos acadêmicos.
Qual método você prefere para escrever os documentos?	Miragem
Qual método de exportação você mais usa?	ABNT
Você acha que o Markers oferece uma experiência mais fácil para o desenvolvimento de artigos acadêmicos e documentos?	Sim, eliminar uma etapa que é a formatação chata do ABNT é uma mão na roda.
O que você acha que pode melhorar na linguagem e em suas aplicações?	Não tenho sugestões no momento.

4 DIFICULDADES E ATUALIZAÇÕES FUTURAS

Uma das maiores dificuldades apresentadas está na interpretação da norma ABNT. Um feedback recebido com frequência é a adição da paginação automática nos sumários e listas de figuras, tabelas e ilustrações. Sua adição simplificaria ainda mais a criação de documentos acadêmicos, visto que a norma ABNT exige que as páginas sejam numeradas e que o sumário e as listas de figuras, tabelas e ilustrações sejam atualizados automaticamente. Provou-se que a adição desta funcionalidade é difícil, visto que - com o atual método e escopo do projeto - seria necessária a criação de uma nova biblioteca JavaScript nativa e diferente dos disponíveis atualmente. Fora testado bibliotecas como o Vivliostyle e Paged.js, nenhum que se mostrou satisfatório para a adição desta funcionalidade. A solução temporária é a adição de um atributo extra na tag `(chap)` e `(img)` para que o usuário possa adicionar a paginação manualmente.

Outros projetos que não foram incluidos neste trabalho de conclusão de curso foi o desenvolvimento de um `plugin` para o editor de texto Visual Studio Code que realiza a

marcação da sintáxe do Markers de forma elegante. Este está disponível para download na *Marketplace* do Visual Studio Code na seguinte URL: https://marketplace.visualstudio.com/items?itemName=mirvoxtm.mks (Acesso em 20 maio 2025). Além do desenvolvimento de uma versão móvel do editor Miragem, que está em desenvolvimento e será disponibilizada no futuro.

Outros feedbacks recebidos consistem na adição de novas tags para a linguagem, estes são possíveis respeitando o escopo do projeto, visto que a linguagem Markers é extensível e permite a adição de novas tags. Porém, por ser um projeto dedicado a ser gerenciado e expandido pela comunidade, a adição de funcionalidades mais recentes foge do escopo deste trabalho de conclusão de curso. Dado isto, será impossível explorar todas as tags possíveis que poderão vir a serem adicionadas, além de futuras versões dos interpretadores e refatoramento nos códigos aqui apresentados.

5 CONCLUSÃO

Como demonstrado pelas entrevistas realizadas e os diversos feedbacks recebidos, a linguagem Markers e suas aplicações são bem aceitas pelos usuários. A linguagem foi recebida com elogios de facilidade de uso e simplicidade tanto por usuários que conheciam linguagens de marcação mais complexas quanto por usuários que nunca haviam utilizado uma linguagem de marcação antes. A aplicação Miragem também foi bem recebida, com usuários elogiando sua interface intuitiva e facilidade de uso. A adição de funcionalidades como o editor online com vários usuários e a possibilidade de salvar documentos na nuvem foram elogiadas.

Os resultados obtidos com os testes de implementação e a construção de exemplos práticos comprovam que o Markers e todo o ecossistema desenvolvido neste trabalho de conclusão de curso atende aos objetivos inicialmente propostos, permitindo a criação de documentos complexos com facilidade e coerência. A capacidade de gerar documentos compatíveis com normas acadêmicas, como a ABNT, reforça seu potencial de aplicação em ambientes educacionais e científicos. Espera-se que a iniciativa deste projeto inspire outros desenvolvedores, usuários e instituições acadêmicas a continuarem explorando e expandindo a linguagem Markers, contribuindo para sua evolução e adoção em larga escala.

REFERÊNCIAS

ABNT - Associação Brasileira de Normas Técnicas. **NBR 14724**, 2002. Disponível em: https://www2.ufjf.br/ppgsaude//files/2008/10/nbr_14724_apresentacao_de_trabalhos.pdf. Acesso em: 14 maio 2025.

Amazon AWS. **O QUE É UM VPS**, 2024. Disponível em: https://aws.amazon.com/pt/what-is/vps/. Acesso em: 13 maio 2025.

ANAND, Rishu. **HOW GOOGLE DOCS HANDLES REAL-TIME COLLABORATION WITH CRDTS**, 2024. Disponível em: https://medium.com/@anand.rishu310/how-google-docs-handles-real-time-collaboration-with-crdts-a7646b5a602b. Acesso em: 20 maio 2025.

Escola Superior de Redes. **CONTAINERS E DOCKER: O QUE SÃO E COMO UTILIZAR**, 2021. Disponível em: https://esr.rnp.br/administracao-de-sistemas/containers-docker/. Acesso em: 13 maio 2025.

Hackage. **MEGAPARSEC: MONADIC PARSER COMBINATORS**, 2024. Disponível em: https://hackage.haskell.org/package/megaparsec. Acesso em: 13 maio 2025.

LBODev. **O QUE É ABSTRACT SYNTAX TREE**, 2024. Disponível em: https://lbodev.com.br/glossario/o-que-e-abstract-syntax-tree/. Acesso em: 08/05/2025.

MALAQUIAS, José Romildo. **PROGRAMAÇÃO FUNCIONAL: PARSERS**, 2025. Disponível em: http://decom.ufop.br/romildo/2012-2/bcc222/slides/14-parsers.pdf. Acesso em: 07 maio 2025.

Postman INC.. **POSTMAN: THE WORLD'S LEADING API PLATFORM**, 2024. Disponível em: https://www.postman.com/. Acesso em: 13 maio 2025.

Professional Advantage. **QUAL A DIFERENÇA ENTRE O MICROSOFT AZURE E O OFFICE 365?**, 2023. Disponível em: https://www.pa.com.au/products/microsoft-office-365/faq/whats-the-difference-between-microsoft-azure-and-office-365/. Acesso em: 07 maio 2025.

SNOYMAN, Michael. **YESOD WEB FRAMEWORK**, 2010. Disponível em: https://www.yesodweb.com/. Acesso em: 13 maio 2025.

SOARES, Alícia. **O QUE É A LINGUAGEM DE MARCAÇÃO, OS SEUS PRINCIPAIS EXEMPLOS E COMO IMPLEMENTÁ-LA?**, 2022. Disponível em: https://voitto.com.br/blog/artigo/linguagem-de-marcacao. Acesso em: 07 maio 2025.

World Wide Web Consortium. **TIM BERNERS-LEE - LONGER BIOGRAPHY**, 2022. Disponível em: https://www.w3.org/People/Berners-Lee/Longer.html. Acesso em: 07 maio 2025.

YJS. **THE LIBRARY FOR REALTIME COLLABORATION**, 2015. Disponível em: https://yjs.dev/. Acesso em: 13 maio 2025.