Documentation MyPandoc

Document Structure Overview

Our Pandoc system uses an Abstract Syntax Tree (AST) to represent structured documents. The AST handles Markdown, JSON, and XML inputs/outputs through a unified representation.

Core Data Types

The AST consists of these primary types:

Document

```haskell
data Document = Document Meta [Block]
```

A document consists of metadata (`Meta`) and a list of content blocks (`[Block]`).

Meta

```haskell
data Meta = Meta
  { metaTitle    :: [Inline]    -- Document title
  , metaAuthors  :: [[Inline]]  -- List of authors
  , metaDate     :: [Inline]    -- Date of the document
  }
```

Metadata includes title, authors (multiple authors possible), and date.

Block

```haskell
data Block
  = Plain [Inline]                          -- Plain text without paragraph
  | Para [Inline]                           -- Paragraph
  | CodeBlock String                        -- Code block
  | RawBlock Format String                  -- Raw block (HTML, LaTeX, etc.)
  | OrderedList ListAttributes [[Block]]    -- Numbered list
  | BulletList [[Block]]                    -- Bullet list
  | DefinitionList [([Inline], [[Block]])]  -- Definition list
  | Header Int [Inline]                     -- Section header
  | Section Int [Inline] [Block]            -- Section with nested blocks
  | Null                                    -- Empty block
```

Block elements represent structural components of a document.

Inline

```haskell
data Inline
  = Str String                              -- Plain text
  | Emph [Inline]                           -- Emphasized text (italic)
  | Strong [Inline]                         -- Strong emphasis (bold)
  | Code String                             -- Inline code
  | RawInline Format String                 -- Raw inline content
  | Link [Inline] Target                    -- Hyperlink
  | Image [Inline] Target                   -- Image
  | Note [Block]                            -- Footnote
  | Span [Inline]                           -- Generic inline container
```

Inline elements represent text-level formatting and entities.

Supporting Types

```haskell
type Target = (String, String)                -- URL and title
newtype Format = Format String                -- Format specifier
type ListAttributes = (Int, ListNumberStyle, ListNumberDelim)
data ListNumberStyle = DefaultStyle | Example | Decimal | LowerRoman | UpperRoman
                     | LowerAlpha | UpperAlpha
data ListNumberDelim = DefaultDelim | Period | OneParen | TwoParens
```

These types provide additional details for specialized elements.

Implementing a New Renderer

To implement a new renderer for the AST, you need to create functions that traverse the AST and generate output in your target format. Here's how to approach it:

1. Create a Main Renderer Function

Start by creating a main function that takes a `Document` and returns a `String`:

```haskell
myFormatRender :: Document -> String
myFormatRender (Document meta blocks) =
  renderMeta meta ++ renderBlocks blocks
```

2. Implement Metadata Rendering

Create a function to render document metadata:

```haskell
renderMeta :: Meta -> String
renderMeta (Meta title authors date) =
  -- Format-specific metadata rendering
```

3. Implement Block Rendering

Create functions to render Block elements:

```haskell
renderBlocks :: [Block] -> String
renderBlocks blocks = concatMap renderBlock blocks

renderBlock :: Block -> String
renderBlock (Para inlines) =
  -- Paragraph rendering
renderBlock (Header level inlines) =
  -- Header rendering
renderBlock (BulletList items) =
  -- Bullet list rendering
-- ... and so on for other block types
```

4. Implement Inline Rendering

Create functions to render Inline elements:

```haskell
renderInlines :: [Inline] -> String
renderInlines inlines = concatMap renderInline inlines

renderInline :: Inline -> String
renderInline (Str text) =
  -- Plain text rendering
```

```
renderInline (Emph inlines) =
  -- Emphasized text rendering
renderInline (Strong inlines) =
  -- Bold text rendering
-- ... and so on for other inline types
```

5. Handle Special Cases

Implement special cases for your format:

```
escapeSpecialChars :: String -> String
escapeSpecialChars = -- Format-specific character escaping
```

Example: JSON Renderer Implementation

The provided `JsonRender` module demonstrates how to implement a renderer:

1. **Main entry point**: `jsonRender` function converts a full Document to JSON
2. **Block rendering**: `jsonRenderBlock` handles each type of Block
3. **Inline rendering**: `jsonRenderInline` and `jsonRenderInlines` handle Inline elements
4. **Helper functions**: For escaping characters, formatting lists, etc.

Key Rendering Patterns

Several patterns can be observed in the JSON renderer that apply to new renderers:

Recursive Rendering

Block elements may contain inline elements, and inline elements may contain other inline elements. Use recursive functions to handle this.

String Escaping

Always handle escaping of special characters required by your output format.

Element Mapping

Map each AST element type to its corresponding representation in your target format.

Composition

Build complex output by composing the output of smaller elements.

Implementation Tips

1. Start with a simple, functional renderer that handles basic elements
2. Add support for more complex elements incrementally
3. Test with simple documents before moving to complex ones
4. Pay attention to whitespace and formatting in your output
5. Consider using helper functions for repeated patterns

Common Challenges

- **Nested formatting**: Elements can be deeply nested, requiring careful recursive handling
- **Special characters**: Different output formats have different character escaping needs
- **Whitespace handling**: Ensuring correct spacing between elements
- **Context-sensitive rendering**: Some elements may need to be rendered differently based on context

By following this structure, you can implement renderers for any desired output format while maintaining a consistent approach across your system.

Parsing Input Formats

In addition to rendering AST to different output formats, Mini Pandoc also needs to parse various input formats (Markdown, XML, JSON) into the AST.

Parser Fundamentals

Our parsing system is built on a basic parser combinator approach:

```haskell
-- Basic parser type from Lib
type Parser a = String -> Maybe (a, String)
```

A parser takes a string input and returns either: - Nothing (if parsing fails) - Just (result, remaining) where result is the parsed value and remaining is the unparsed portion of the input

Example: XML Parser Implementation

The provided XML parser demonstrates the approach:

1. **Parser structure definition**:

   ```haskell
   data XmlValue
     = XmlElement String Int [(String, String)] [XmlValue]
     | XmlText String
   ```

2. **Basic parsers** for elements, text, and attributes:

   ```haskell
   parseXmlValue :: Int -> Parser XmlValue
   parseXmlText :: Parser XmlValue
   parseXmlElement :: Int -> Parser XmlValue
   parseXmlAttribute :: Parser (String, String)
   ```

3. **Conversion functions** that transform the parsed structure to AST:

   ```haskell
   xmlToDocument :: XmlValue -> Document
   getmeta :: [XmlValue] -> Meta
   getblocks :: [XmlValue] -> [Block]
   getblock :: XmlValue -> [Block]
   getInline :: XmlValue -> [Inline]
   ```

4. **Helper functions** for finding elements and extracting content:

   ```haskell
   findElement :: String -> [XmlValue] -> Maybe XmlValue
   extractText :: String -> [XmlValue] -> Maybe String
   ```

5. **Entry point function** that handles the entire parsing process:

   ```haskell
   parsexml :: String -> Document -> Maybe Document
   ```

Implementing a New Parser

To create a new parser for an input format, follow these steps:

1. Define an intermediate representation Create data types that represent the structure of your input format:

```haskell
data MyFormatValue
  = MyElement String [MyFormatValue]
  | MyText String
  -- Other format-specific elements
```

2. Create basic parsers Implement parsers for each element in your format:

```haskell
parseMyFormat :: Parser MyFormatValue
parseMyFormat = parseMyElement <|> parseMyText

parseMyElement :: Parser MyFormatValue
parseMyElement = do
  -- Format-specific parsing logic

parseMyText :: Parser MyFormatValue
parseMyText = do
  -- Text parsing logic
```

3. Implement conversion to AST Create functions to convert your parsed structure to the AST:

```haskell
myFormatToDocument :: MyFormatValue -> Document
myFormatToDocument value =
  Document meta blocks
  where
    meta = extractMeta value
    blocks = extractBlocks value

extractMeta :: MyFormatValue -> Meta
extractMeta = -- Logic to extract metadata

extractBlocks :: MyFormatValue -> [Block]
extractBlocks = -- Logic to extract blocks
```

4. Create helper functions Implement any helper functions needed for extraction and conversion:

```haskell
findMyElement :: String -> [MyFormatValue] -> Maybe MyFormatValue
extractMyText :: MyFormatValue -> String
```

5. Create the main entry point Provide a function that parses input text and returns a Document:

```haskell
parseMyFormat :: String -> Document -> Maybe Document
parseMyFormat input _ = do
  value <- runMyFormatParser input
  return (myFormatToDocument value)
```

Parser Combinator Patterns

Several key patterns are used in the parsing system:

1. **Alternative (<|>)**: Try one parser, and if it fails, try another

   ```haskell
   parseXmlValue level = parseXmlElement level <|> parseXmlText
   ```

2. **Sequential composition (do notation)**: Run parsers in sequence

   ```haskell
   parseXmlElement level = do
     _ <- parseChar '<'
   ```

```
name <- parseSome (parseAnyChar (['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9']))
-- other parsing steps
```

3. **Repetition (parseMany, parseSome)**: Parse repeated patterns

   ```
   attribute <- parseMany parseXmlAttribute
   ```

4. **Character and string matching**:

   ```
   _ <- parseChar '<'
   _ <- parseString "</"
   ```

Implementation Tips for Parsers

1. **Start small**: Begin with basic elements and add complexity gradually
2. **Test frequently**: Test each parser function individually before combining
3. **Handle edge cases**: Consider empty documents, malformed input, etc.
4. **Use meaningful error messages**: When possible, provide context for parsing failures
5. **Optimize for readability**: Parser code should be clear and maintainable

By following these patterns, you can implement parsers for various input formats that convert content into your AST for further processing and rendering.