PANDOC

POWERFUL AND NIMBLE DOCUMENT CONVERTER

PANDOC

Preliminaries

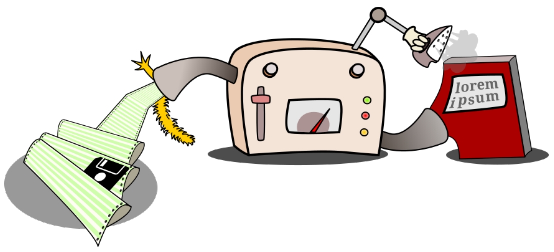
binary name: mypandoc
language: Haskell
compilation: stack wrapped in a Makefile *(see below)*

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,...), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Pandoc is a free and open-source document converter that supports numerous text file formats. It was developed by John MacFarlane and is written in Haskell. Pandoc enables conversion between various file types, thereby facilitating document exchange across different applications and platforms. It is widely used by the scientific community and is integrated into many software tools.

https://pandoc.org/

This project is a simplified version of Pandoc. The goal is to implement a program that can interact with documents. The program takes a document as input and outputs the document in another format.



Document

A **document** can be described as an organized structure of information, defined by a set of syntactic rules. This structure comprises two main parts: the **header** and the **content**.

1. **Header:**
    - ✓ **Title**: A document have a title.
    - ✓ **Author (Optional)**: An optional field to indicate the document's author.
    - ✓ **Date (Optional)**: An optional field to specify the creation date.
2. **Content:**
    - ✓ The main body of the document is made up of various elements:
        - **Text**: A sequence of ASCII characters.
        - **Formatting:**
            - **Italic**.
            - **Bold**.
            - **Code**.
        - **Links and Images:**
            - **Link**: Composed of text and additional content.
            - **Image**: Composed of text and additional content.
        - **Structural Elements:**
            - **Paragraph**: A grouping of content elements forming a paragraph.
            - **Section**: A section with an optional title, containing more content.
            - **Code Block**: A block of code.
        - **List Elements:**
            - **List**: Composed of items.
            - **Item**: Represents an individual item in a list, composed of various content elements.

The document, according to this structure, combines the header and the content, where the content can be a combination of text, formatting, links, images, structural elements, and lists.

Formats

Your program could handle different formats, but for this project we will focus on three formats: ***XML***, ***JSON*** and ***Markdown***.

Every format will have its own way to represent a **Document**.

We don't expect you to implement the full specifications of these formats, but only the parts that are relevant to this project.

For more information, you can refer to the examples given in the **examples** folder provided with the subject.

XML

Extensible Markup Language (**XML**) is a markup language and file format for storing, transmitting, and reconstructing arbitrary data.
https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction

```xml
<document>
  <header title="Simple example"></header>
  <body>
    <paragraph>This is a simple example</paragraph>
  </body>
</document>
```

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format which is a subset of the Javascript language.
https://www.json.org/json-en.html

```json
{
  "header": {
      "title": "Simple example"
  },
  "body": [
    "This is a simple example"
  ]
}
```

MARKDOWN

Markdown is a lightweight markup language for creating formatted text using a plain-text editor.
https://www.markdownguide.org/basic-syntax

```
---
title: Simple example
---

This is a simple example
```

Parser

For this project you will have to implement a **parsing library**, which will help you to implement the parsers for the different file formats.

In the Bootstrap, your going to see how to implement a parsing library in Haskel. You must use it in your project.

> The more time you spend on it, the simpler the project will be.

Your program must use your library to parse the input file.

Feel free to add any feature that you think is relevant to your parsing library.

> You are not allowed to use any parsing library other than yours.
> This point will be checked during the defense of your project.

Your parser is going to transform the input file into a **Document**.

Run the program

Your program must support the following options:

- ✓ -i : path to input file (mandatory)
- ✓ -f : format of the output (mandatory)
- ✓ -o : path to the output file
- ✓ -e : format of the input file

If no option or invalid options are provided your program must return **84** and display a usage message (eventually accompagned with an explicite error message of your choice)

If no output file is provided, the program must output the result on the standard output.
If no input format is provided, the program must detect the format.

```
Terminal
~/B-FUN-400> ./mypandoc
USAGE: ./mypandoc -i ifile -f oformat [-o ofile] [-e iformat]

    ifile     path to the file to convert
    oformat     output format (xml, json, markdown)
    ofile     path to the output file
    iformat     input format (xml, json, markdown)
```

Example:

```
Terminal
~/B-FUN-400> cat "example/example.xml"
<document>
  <header title="Simple example"></header>
  <body>
    <paragraph>This is a simple example</paragraph>
  </body>
</document>
~/B-FUN-400> ./mypandoc -f markdown -i "example/example.xml" | cat -e
---$
title: Simple example$
---$
$
This is a simple example$
```

Implementation

Your project is going to be evaluated with **automated tests** and **defense**.

To be considered a valid project, it is essential that your code adheres to the following requirements:

- ✓ Your project must use your parsing library.
- ✓ Support XML and JSON formats as input is a prerequisite for your project.
- ✓ Handling XML, JSON, and Markdown formats for output.

Once you have done the basic requirements, you can handle Markdown as input.

During the defense, we are going to evaluate the **quality of your code** and the **architecture of your project**.

Think about how you handle `Document`, how you handle the different formats, options, errors, parsing.

You can write documentation about your `Document` and Unit test to have a very nice project.

Bonus

Feel free to add more features to your project, but don't forget to mention them in your README.md

More formats, more options, more features, more tests, more documentation, ...

Build with Stack

Stack is a convenient build tool/package manager for Haskell.
Its use is required for this project, with **version 2.1.3 at least**.#br

It wraps a build tool, either **Cabal** or **hpack**.
You are required to use the hpack variant (package.yaml file in your project, autogenerated .cabal file).

This is what stack generates by default with `stack new`.

Stack is based on a package repository, **stackage**, that provides consistent snapshots of packages.
The version you use must be in the **LTS 23** series (`resolver: 'lts-23.3'` in stack.yaml).

In `stack.yaml`, extra-dependencies cannot be used.

base, **optparse-applicative** and **containers** are the only dependencies allowed in the `lib` and `executable` sections of your project (package.yaml).
There is no restriction on the dependencies of the `tests` sections.

You must provide a **Makefile** that builds your stack project (i.e. it should at some point call 'stack build').

'stack build' puts your executable in a directory that is **system-dependent**, which you may want to copy.
A useful command to learn this path in a system-**independent** way is:
`stack path --local-install-root`.

The automatic test system expects to find the file stack.yaml of your project at the root of your repository v 1.1