Elements of Programming Languages
Coursework 2
Version 1.8 (Updated November 19, 2015)
Due: November 23, 2015, 4pm

Overview

This coursework assignment asks you to combine several concepts covered in the course so far, to implement a simple *domain specific language*. The language a lightweight *markdown language*. "Markdown" is a human-readable format for hypertext, ofren used in Wiki pages. There are a number of different syntaxes for markdown, and we will consider one with a core subset of features that make it easy to translate into two other domain-specific languages: HTML (used in Web browsers) and LaTeX (used for more professional / technical document preparation.)

Although it is relatively straightforward to translate a markdown language to HTML or LaTeX text directly, we ask you instead to do this via a *pretty-printing* library, which is essentially a simple domain-specific language embedded in Scala. One advantage of this approach is that it separates out some low-level concerns (such as dealing with indentation of nested structures properly) from the high-level translation. You will also implement the pretty-printing DSL itself.

Finally, you will also implement a random generator for markdown documents that can be used to test the "correctness" of your translations under different circumstances (e.g. by rendering the same document in different formats and comparing). Again, while it is possible particularly difficult to generate random documents directly, we ask you to do so using a *random generation* library, which is another example of a domain-specific language embedded in Scala. You will complete the implementation of this DSL.

In this assignment, you may use Scala standard library functions and any features of Scala. The assignment can be completed using concepts covered up to lecture 11 and does not rely on mutable data structures or imperative programming (although it may be convenient to use them).

A simple markdown language

The simple markdown language used in this assignment is called MiniMD. We provide a parser from MiniMD to Scala-based ASTs. The basic features of MiniMD (illustrated using examples) are:

- Free text with bold, italic and underline syntax

```
Some text.
*Bold*, `italic`
and _underlined_
```

- Numbered and bulleted lists (without nesting)

```
Shopping list:

* Apples
* Oranges

To-do list:

1. Wash dishes
1. Brush teeth
```

Note that the exact numbers used in the numbered list syntax do not matter, they are automatically renumbered by the renderer.

- Section and subsection headings:

```
== Section ==

Roses are red.

=== Subsection ===

Violets are blue.
```

- Unformatted text:

```
{{{
*Bold*, `italic`
and _underlined_
markup symbols
are ignored.

* So are lists

1. and so on.
}}}
```

- Link syntax

```
(Link)[http://google.com]
```

We provide some code in `CW2.scala` that provides a framework including ASTs for MiniMD, plus code for parsing MiniMD text into ASTs — this turns out to be surprisingly tricky.

The first goal of this assignment is to implement formatters that "translate" MiniMD ASTs back to the concrete syntax outlined above, as well as to two other languages: HTML (which can then be displayed in a browser), and LaTeX (which can then be compiled into a PDF or other document formats). In both cases, the structure and content should be printed in a user-readable way, using indentation and line breaks appropriately.

The second goal of this assignment is to implement a random data generator for testing the MiniMD formatters.

A subset of HTML

We will map MiniMD to a subset of HTML (the HyperText Markup Language) which is used for Web pages. HTML is widely documented on the Web, for example, this tutorial.

The basic element of HTML markup is the *tag*, a matched pair of tokens surrounded in angle brackets. The opening tag is a single word surrounded by `< >` and the closing tag is the same word surrounded by `</ >`. The text between the two matching tags is called the *content* of the tag. For example, `<b>abc</b>` consists of an opening `<b>` tag, the string `abc` as content, and a closing `</b>` tag. In HTML, `b` stands for bold, so this text would be rendered **abc** in a browser. Additional tags for text markup include `<p>` (paragraph), `<i>` (italic) and `<u>` (underline).

Tags can optionally have arguments consisting of attribute-value pairs, written `attr="value"`. For example, the `<a>` (anchor) tag can have an attribute called `href` whose value is a URL. This indicates that the text inside the `<a>` tag should be treated as a clickable link to the given URL. So, for example, the HTML tag

```
<a href="http://www.inf.ed.ac.uk/teaching/courses/epl/">course website</a>
```

corresponds to a clickable link to the course website.

In HTML, spaces or newlines separate words and tags, but multiple spaces or newlines are usually treated the same as a single space. However, the `<pre>` tag suspends markup; its textual content is rendered in a monospace font and whitespace is significant. In HTML, comments are started by `<!--` and ended by `-->`.

```
<html>
 <body>
 <!-- Beginning of MiniMD code -->
 <p>Some text.
  <b>Bold</b>,
  <i>italic</i>
  and <u>underlined</u></p>
 <p>Shopping list:</p>
 <ul>
   <li> Apples </li>
   <li> Oranges </li>
 </ul>
 <p>To-do list:</p>
 <ol>
 <li> Wash dishes </li>
 <li> Brush teeth </li></ol>
 <h1>Section</h1>

 Roses are red.

 <h2>Subsection</h2>

 Violets are blue.
<pre>
*Bold*, `italic` and _underlined_
markup symbols are ignored here.

* So are lists

1. and so on.
</pre>
<a href="http://google.com">Link</a>
<!-- End of MiniMD code -->
</body>
</html>
```

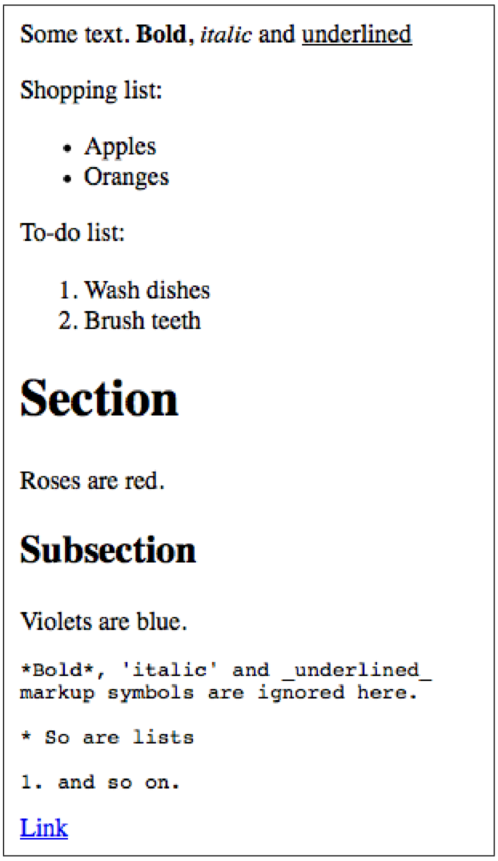

Figure 1: Example translated to HTML, and rendered by `firefox`

Bulleted and numbered lists are introduced using the `<ul>` and `<ol>` tags respectively. In these tags, list items are marked up using `<li>`.

Table 1 summarises the mapping from MiniMD constructs to HTML. Note that free text blocks should be wrapped in the paragraph tag `<p>...</p>`.

A template HTML file `template.html` (shown in Figure 3) deals with the boilerplate that needs to go into any HTML page, such as the `html`, `title` and `body` tags. Figure 1 illustrates the examples above rendered in HTML.

A subset of LaTeX

LaTeX is a document preparation system, popular for scientific and technical writing due to its strong support for mathematical notation. LaTeX and HTML have many differences, but there is a core subset of features that are similar in both languages. More details can be found here: http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/

In LaTeX, text is marked up using *macros* and *environments*. A macro looks like this: `\macro{argument}`. For example, the `\textit`, `\textbf`, and `\underline` are macros for italic, bold and underlined text respectively. Section and subsection headings are marked up using the `\section` and `\subsection` macros. Macros can also take more than one argument (or no arguments). The `\href{url}{text}` macro, which we will use for links, takes two arguments, a URL and the text that should actually appear in the document.

In LaTeX, *environments* are delimited by matched pairs of macros as follows:

```
\documentclass{article}
\usepackage[colorlinks=true]{hyperref}
\begin{document}
% Beginning of MiniMD code
Some text.
\textbf{Bold}, \textit{italic}
and \underline{underlined}

Shopping list:
\begin{itemize}
\item Apples
\item Oranges
\end{itemize}

To-do list:
\begin{enumerate}
\item Wash the dishes
\item Brush teeth
\end{enumerate}

\section{Section}

Roses are red.

\subsection{Subsection}

Violets are blue.
\begin{verbatim}
*Bold*, `italic` and _underlined_
markup symbols are ignored here.

* So are lists

1. and so on.
\end{verbatim}
\href{http://google.com}{Link}
% End of MiniMD code
\end{document}
```

Some text. **Bold**, *italic* and <u>underlined</u>
Shopping list:

- Apples
- Oranges

To-do list:

1. Wash the dishes
2. Brush teeth

1 Section

Roses are red.

1.1 Subsection

Violets are blue.

```
*Bold*, 'italic' and _underlined_
markup symbols are ignored here.

* So are lists

1. and so on.
```

Link

Figure 2: Example translated to LaTeX, and rendered by `pdflatex`

```
\begin{environment}
\end{environment}
```

The environments needed for this assignment are `document` (which delimits the whole document), `itemize` (bulleted list), `enumerate` (numbered list), and `verbatim` (unformatted text). The `\item` macro takes no arguments, and indicates the beginning of an item in a bulleted or numbered list environment.

In LaTeX, there is no need to explicitly delimit blocks of text; instead, paragraphs are separated by a completely blank line. Spaces are needed to separate words, but two or more spaces are treated the same as one space; similarly, multiple consecutive blank lines are treated the same as one blank line. Line breaks can also be added explicitly using the `\\` symbol, but you should not need to do this.

In LaTeX, the `%` symbol starts a comment, which extends to the rest of that line.

Table 2 illustrates the mapping from MiniMD to LaTeX. Figure 4 shows the LaTeX template that needs to be used around the generated code; it includes the `hyperref` package that provides the `href` macro.

Figure 2 shows how the above constructs would be rendered by LaTeX into a PDF using the `pdflatex` program. You can test this behaviour yourself by running the command `pdflatex` command on a generated LaTeX file from DICE.

Objectives

The provided code file `CW2.scala` defines the abstract syntax of MiniMD as well as a parser and pretty-printer for it (the latter using the `Printer` interface, which you need to implement.) We also provide a sample solution `CW2Solution.jar` and some example MiniMD inputs.

The `CW2.scala` file can be loaded into Scala as follows:

```
scala> :load CW2.scala
```

In addition, both programs can be run by providing them as arguments to Scala:

```
scala CW2.scala <infile> <mode> <outfile>
scala CW2Solution.jar <infile> <mode> <outfile>
```

Here, `<infile>` is an input file name, or `RANDOM` to indicate that the input should gbe generated randomly. The `<mode>` is one of `md`, `latex` or `html` and is used to choose which format to use for the results. Finally, `<outfile>` is an optional output file name; if omitted, the output is just printed to the screen.

Finally, it is also possible to place `CW2Solution.jar` on the Scala classpath so that you can interactively explore its behaviour, as follows:

```
scala -cp CW2Solution.jar
scala> import CW2._
```

In addition, we provide a file implementing some test cases for the printer DSL called `TestCases.scala`. You can run this using the sample solution as follows:

```
scala -cp CW2Solution.jar TestCases.scala
```

or using your solution as follows:

```
scala -i CW2.scala TestCases.scala
```

You do not need to worry about dealing with ("escaping") LaTeX or HTML special characters embedded in the AST strings, as in the following MiniMD document:

```
This will \emph{render} <b>differently</b> in
<em>HTML</em> and \textbf{LaTeX}!
```

This is a legitimate problem that would need to be solved in a full-scale MiniMD rendering engine, but it is not particularly significant from our point of view and we will not deduct credit for failure to handle such scenarios correctly.

You also have some freedom concerning where to insert spacing and newlines. The exact formatting of the HTML or LaTeX output of formatters doesn't matter very much, as long as the end product (rendered by a browser or converted to PDF by LaTeX) is reasonable.

The rest of this handout defines exercises for you to complete, building on the partial implementation in `CW2.scala`. You may add your own function definitions or other code, but please use the given templates for the functions we ask you to write in the exercises, to simplify automated testing we may do. Also, please do not change code in the `CW2.MiniMDParser` and `CW2.Main` submodules.

This assignment relies on material covered up to Lecture 12 (November 5). The two sections of this assignment are independent and can be attempted in any order. Partial credit may be given for progress on each part (including for commented-out code if it demonstrates progress towards a solution).

This assignment is graded on a scale of 30 points, and amounts to 15% of your final grade for this course. You may not work in groups on this assignment and must document any meaningful discussions you have about this assignment (or external sources you consult) in the process of constructing your solutions.

Submission instructions You should submit a single file, called `CW2.scala`, with missing code filled in as specified in the exercises in the rest of this handout. To submit, use the following DICE command:

```
$ submit epl 2 CW2.scala
```

The submission deadline is 4pm on Monday, November 23.

Warning! Some students have reported problems running

1 Pretty-printing

The main goal of this assignment is to provide two implementations of MiniMD: one that renders the markdown file as an HTML snippet (which can be rendered in a browser, if it is merged into the provided HTML template), and another that renders it as a suitable LaTeX document (which can in turn be compiled into a PDF or other document formats by the `pdflatex` command). In both cases, the structure and content should be handled appropriately.

You will implement these translations by defining translations from MiniMD to a domain-specific language (DSL) for *pretty-printing*. You will also implement this DSL.

The interface of the pretty-printing DSL is as follows:

```
trait Printer {
  type Doc
  def text(s: String): Doc
  def line: Doc
  def nil: Doc
  def append(x: Doc, y: Doc): Doc
  def nest(level: Int, doc: Doc): Doc
  def unnest(doc: Doc): Doc
  def print(doc: Doc): String
}
```

The interface refers to an abstract type `Doc` that you will also need to implement. The operations are:

1. `text(s)`, a document containing some text `s: String`.
2. `line`, a document representing a newline followed by a the number of spaces specified by the current nesting level (if nesting is currently in effect)
3. `nil`, an empty document.
4. `append(d1,d2)` which sequentially combines two documents `d1` followed by `d2`. Note that `d1 <> d2` is an abbreviation for `append(d1,d2)`.
5. `nest(i, d)`, a document that prints sub-document `d` at a given relative indentation level
6. `unnest(d)`, which temporarily suspends indentation while printing `d`
7. `print(d)`, which takes a document and renders it as a string.

For example, the following code builds a document value:

```
text("a") <> nest(2, line <> text("b")) <> line <> text("c")
```

which, when `print`ed, should yield:

```
a
  b
c
```

In addition, to pretty-print a Scala function definition we might define a `Doc` value as follows:

```
text("def␣f(x:␣Int)␣=␣{") <>
nest(2, line <> text("if␣(x␣<␣1)␣{") <>
  nest(2, line <> text("println(x␣+␣1)")) <> line <> text("}")) <>
line <> text("}")
```

which should result in the following output string when we call `print`:

```
def f(x: Int) = {
  if (x < 1) {
    println(x + 1)
  }
}
```

To implement this DSL, you will need to implement an **`object`** or **`class`** that extends the `Printer` trait. The main challenge is to deal with the *stateful* behaviour of indentation, so that documents spanning multiple lines are printed correctly within a `nest`, and so that `unnest` temporarily suspends indentation. There are several ways to do this, and some suggestions are given below.

1.1 Simple pretty-printing examples

First, you will write some simple pretty-printing operations that extend the `Printer` trait.

```
def quote: Doc -> Doc
def braces: Doc -> Doc
def anglebrackets: Doc -> Doc
```

Their behaviour is as follows: `quote(doc)` surrounds `doc` with double quotation marks `"`, `braces` surrounds `doc` with left and right braces {, }, and `anglebrackets(doc)` surrounds `doc` with left and right angle brackets <, >.

Exercise 1. *Define the `quote`, `braces` and `anglebrackets` operations in terms of other operations of the `Printer` trait.*

[2 marks]

Next, you will define a *combinator* called `sep` such that `sep(d,l)` constructs a document in which consecutive elements of `l` are separated by `d`. (If `l` is empty then the resulting document is empty; if there is only one element then the resulting document is that element and `sep` is unused.)

Exercise 2. *Define the separation combinator `sep:(Doc,List[Doc]) -> Doc`*

[3 marks]

1.2 Implement the Printing Language

In this section you are asked to provide an implementation of the printing language. To get you started, we suggest two possibilities, one based on mutable state and another following a pure, functional approach.

In the first approach, `Doc` is defined as an abstract syntax tree type `DocAST`:

```
abstract class DocAST
```

with case subclasses for each of the `Doc` operations in the `Printer` interface. We can then define an object `StatefulPrinter` in which the language operations in the interface simply generate the corresponding AST nodes, while the `print` operation traverses the AST. For the `print` operation, `StatefulPrinter` should create an instance of an auxiliary `PrinterState` class that traverses a `DocAST` while maintaining some state (using mutable **`var`** fields) for the printing process, such as the indentation level, whether an `unnest` block has been entered, as well as a `String` containing the output so far. When traversing the `DocAST` data structure, the state should be updated appropriately, and the indentation level and `unnest` status should be taken into account when processing `Line` nodes, so that appropriate indentation is (or isn't) produced.

The second approach is based on a purely functional implementation that maintains the printing state in a different way: `Doc` is defined as the following function type

```
type FDoc = (Int, Boolean) => String
```

Here, the `Int` component represents the current indentation level, the `Boolean` component represents whether indentation is suspended, and the `String` component represents the string produced.

In this functional approach, we will define **`object`** `PurePrinter` in which `Doc` is the type `FDoc` above, and the `Doc` operations are implemented using this type. Moreover, the `print` operation applies its `FDoc` argument to sensible initial values for the indentation level, unnest status, and output string.

Exercise 3. *Define a working implementation `MyPrinter` of `Printer`. You may follow either of the strategies outlined above, but any correct implementation is acceptable.*

[5 marks]

1.3 Defining the translations

Finally, you will define translations from `MiniMDExpr` ASTs to the `Doc` type chosen in the previous part. To do this, we provide a trait `Formatter[T]` defined as follows:

```
trait Formatter[T] {
  def format(e: T): Doc
  def formatList(xs: List[T]): Doc = sep(nil,xs.map{x: T => format(x)})
}
```

Note that `format` is abstract, while `formatList` is defined in terms of `format` and `sep`, so you can automatically use it in any class or object extending `Formatter[T]` (even in the definition of `format`).

We suggest you import `MyPrinter` at this point, so that the `Printer` methods and `Doc` type can be used without the explicit prefix notation.

Exercise 4. *Define a formatter* `MarkdownFormatter` *that extends* `Formatter[MiniMDExpr]` *by implementing the* `format` *method so as to reconstruct the MiniMD source text for a given* `MiniMDExpr`*. For this task, the* `nest` *and* `unnest` *operations should not be needed.*

[2 marks]

Exercise 5. *Define a printer* `LatexFormatter` *that converts a MiniMD expression to a document consisting of equivalent LaTeX text. For readability, the document should employ newlines to separate paragraph blocks and list items, and use* `nest` *to indent content inside environments. However, indentation should be suspended inside a* `verbatim` *environment.*

[3 marks]

Exercise 6. *Define a printer* `HtmlFormatter` *that converts a MiniMD expression to a document consisting of HTML text. For readability, the document should employ newlines to separate paragraph blocks, section tags, and list items, and use* `nest` *to indent content inside list tags. However, indentation should be suspended in the* `<pre>` *tag.*

[3 marks]

In both printers, you may find it helpful to define additional HTML or LaTeX-specific helper operations to simplify the specification of the printer.

2 Random data generation and testing

In this part, you will implement a small library for random generation of data structures, to generate sample input MiniMD files that you can use to test your code.

The interface to the random data generation DSL is:

```
trait Gen[+A] {
  val rng = scala.util.Random
  def get(): A // abstract
  def map[B]( f: A => B): Gen[B] = {
    val g = this;
    new Gen[B] {
      def get() = f(g.get())
    }
  }
  def flatMap[B](f: A => Gen[B]): Gen[B] = ...
}
```

An instance of `Gen[A]` provides a `get` method that should generate a random `A`-value. In addition, `Gen[A]` provides two functions `map` and `flatMap`. The `map` function (whose definition is shown above) allows us to change the return value of a generator by applying a function to it, while the `flatMap` function allows us to chain generators.

For convenience, `Gen[A]` also maintains a local reference `rng` to Scala's `scala.util.Random` object, which provides several useful methods, such as:

```
def nextBoolean: Boolean // a random Boolean
def nextInt(n:Int) => Int // a random number between 0 and n-1
```

In addition, the `Rng` object provides some useful operations on generators:

```
def const[T](c: T): Gen[T]
def flip: Gen[Boolean]
def range(min: Integer, max: Integer): Gen[Integer]
def fromList[T](items: List[T]): Gen[T]
```

The `const(c)` generator always generates `c`. The `flip` generator "flips a coin" to generate a Boolean value, that is, it generates `true` with probability 0.5 and `false` otherwise. The `range` generator chooses a number uniformly at random between `min` and `max` (including both endpoints as possibilities). So for example, `range(1,5)` chooses among 1, 2, 3, 4, and 5 with each choice being equally likely at probability 0.2. Finally, `fromList` chooses among the elements of a (nonempty) list, uniformly at random. Hence, `fromList(List(1,2,3,4,5))` has the same behaviour as `range(1,5)`.

The `Gen` interface implements the `map` and `flatMap` methods, which means that we can use Scala's **`for`** comprehension notation to work with generators. This is especially handy for building generators whose arguments are provided by other generators. For example, to define a generator of random pairs (i, j) where i is between 1 and 10 and j is between 1 and i, we can do this:

```
for(i <- range(1,10); j <- range(1,i)) yield (i,j)
```

Similarly, to define a generator that generates a random number between 1 and 5 and then generates a list of that length, each of whose entries are obtained by coin flips, we can do this:

```
for(i <- range(1,5) ;
    l <- genList(i, flip)) yield l
```

2.1 Implement the random generator DSL

You will first implement the `Gen` interface itself. The first task is to fill in the definitions of the basic generators above, which will be useful in defining more interesting generators later. One is already done for you to illustrate how we can define a value of type `Gen[T]` using an anonymous object definition:

```
def flip: Gen[Boolean] = new Gen[Boolean] {
  def get() = rng.nextBoolean()
}
```

Exercise 7. *Implement the remaining basic generators, specifically `const`, `range` and `fromList`. For these implementations you may use methods of `scala.util.Random`.*

[3 marks]

You should not need to use methods from `scala.util.Random` from now on.

Next, notice that `Gen` is a trait that provides an abstract method `get`, and also includes two methods `map` and `flatMap`. Since these methods are present, we will be able to use the convenient *for-comprehension* syntax with `Gen`. The `map` operation is implemented for you, but the `flatMap` operation is not. You need to implement it. In doing so, first make sure you understand the definition of `map`, and in particular the reason why we locally bind **`this`** to variable `g` instead of using **`this`** inside the anonymous definition of **`new`** `Gen[B]`.

Exercise 8. *Implement the `flatMap` operation of `Gen`.*

[2 marks]

Next you need to implement some additional operations that should be helpful in defining data generators.

Exercise 9. *Define generator* `genFromList: List[Gen[A]] => Gen[A]` *such that* `genFromList(gs)` *chooses a generator randomly from the input list and uses it to generate a random* `A` *value.*

[1 mark]

Exercise 10. *Define generator* `genList: (Int,Gen[A]) => Gen[List[A]]` *such that* `genList(n,g)` *generates a list of* `n` *items, each generated randomly using generator* `g`*.*

[2 marks]

2.2 Define random generators

In this section you will use the `Gen` interface, specified above, to construct some generators. You should use the primitive combinators and should not directly call methods of `scala.util.Random` in the exercises in this section.

The following table lists some test data values for section, subsection, list item, link and free text values.

section	`"Chapter␣1","Introduction","Conclusion"`
subsection	`"Section␣1.1","Table␣of␣Contents","References"`
listitem	`"Apples","Oranges","Spaceship"`
link	`("Google","http://www.google.com"),("Facebook","http://www.facebook.com")`
unformatted	`"==␣This␣isn't␣valid␣MiniMD␣===\n","*Neither_␣_is'␣'this*\n"`
freetext	`"It␣was␣a␣dark␣and␣stormy␣night","It␣was␣the␣best␣of␣times"`

Exercise 11. *Define generators*

```
def genSectionText: Gen[String]
def genSubsectionText: Gen[String]
def genListItemText: Gen[String]
def genLinkText: Gen[(String,String)]
def genVerbatimText: Gen[String]
def genFreeText: Gen[String]
```

that build random text generators using the values above.

[2 marks]

Next, we specify how to generate a test MiniMD AST. Given a size parameter `n`, we want to construct an AST consisting of a single `MDDoc` containing `n` elements. These elements should be selected (uniformly) randomly from among the following 6 choices:

```
MDPar MDBulletedList MDNumberedList MDSectionHeader MDSubsectionHeader MDVerbatim
```

If MDPar is generated, then the content should be 3–5 randomly generated instances of the following types:

```
MDFreeText MDBold MDItalic MDUnderlined MDLink
```

If a free text, bold, italic, or underlined text element is generated, its text should be provided by the `genFreeText` generator. Similarly, the text for an unformatted element or link should be generated by the corresponding generator.

Likewise, the section and subsection text should be generated by the respective generators `genSectionText` and `genSubsectionText`. For a link element, the link text should be generated by taking the first component of `genLinkText` and the URL should be taken from the second element.

In addition, if a bulleted list or numbered list is generated, it should have 2–4 elements (chosen randomly) and each of these elements should be consist of a single free text node with content generated by `genListitemText`.

Exercise 12. *Using the combinators in the* `Gen` *interface, write a generator* `genMiniMDExpr: Int => Gen[MiniMDExpr]` *for* `MiniMDExpr`.

[4 marks]

A Change log

- Version 1.1 (November 1, 2015)
  - Fixed incorrect handling of nesting in LatexFormatter in sample solution.
- Version 1.2 (November 9, 2015)
  - Clarified how link text is to be generated
  - Fixed typo: MDUnformattedText should be MDVerbatim
  - Clarified how `Doc` values are used and printed
- Version 1.3 (November 10, 2015)
  - Fixed mistake in the document example, added another (smaller) example of use of nesting
  - Clarified description of the `line`
- Version 1.4 (November 11, 2015)
  - Added suggestion for how to interactively use the sample solution
  - Added `TestCases.scala` containing tests of the pretty-printing operations
- Version 1.5 (November 16, 2015)
  - Bugfix to `CW2Solution.jar` to avoid extra newlines in markdown formatting of `MDVerbatim` and LaTeX formatting of enumerated lists.
  - Fixed instructions for running `TestCases.scala`
  - Fixed numbering of exercises 7/8
  - Clarified that there appears to be an incompatiblity on Windows and code should be tested on DICE
- Version 1.6 (November 16, 2015)
  - Improved instructions for running test suite
  - Fixed usage information in CW2.scala
  - Fixed error in random generation for verbatim blocks
  - Fixed numbering of exercises 9–12 in CW2.scala
- Version 1.7 (November 17, 2015)
  - Fixed minor bug in random generation (off by one error in uses some of range) in sample solution.
- Version 1.8 (November 19, 2015)
  - Fixed typo in comprehension example

Free text and highlighting	Some text. *Bold*, `italic` and _underlined_	Some text. Bold, <i>italic</i> and <u>underlined</u>
Lists	Shopping list: * Apples * Oranges To-do list: 1. Wash dishes 1. Brush teeth	Shopping list: - Apples - Oranges To-do list: - Wash dishes - Brush teeth
Sections	== Section == Roses are red. === Subsection === Violets are blue.	Section Roses are red. Subsection Violets are blue.

Unformatted text	{{{
	Bold, `italic`
	and _underlined_
	markup symbols
	are ignored.
	
	* So are lists
	
	1. and so on.
	}}}
		<pre>
		Bold, `italic`
		and _underlined_
		markup symbols
		are ignored.
		
		* So are lists
		
		1. and so on.
		</pre>

Links	(Link)[http://google.com]	Link

Table 1: Mapping from MiniMD to HTML

```
<html>
 <body>
 <!-- Beginning of MiniMD code -->
 <!-- End of MiniMD code -->
 </body>
</html>
```

Figure 3: HTML template

Free text and highlighting	Some text. *Bold*, `italic` and _underlined_	Some text. \textbf{Bold}, \textit{italic} and \underline{underlined}
Lists	Shopping list: * Apples * Oranges To-do list: 1. Wash dishes 1. Brush teeth	Shopping list: \begin{itemize} \item Apples ... \item Oranges \end{itemize} To-do list: \begin{enumerate} \item Wash the dishes ... \item Brush teeth \end{enumerate}
Sections	== Section == Roses are red. === Subsection === Violets are blue.	\section{Section} Roses are red. \subsection{Subsection} Violets are blue.
Unformatted text	{{{ *Bold*, `italic` and _underlined_ markup symbols are ignored. * So are lists 1. and so on. }}}	\begin{verbatim} *Bold*, `italic` and _underlined_ markup symbols are ignored. * So are lists 1. and so on. \end{verbatim}
Links	(Link)[http://google.com]	\href{http://google.com}{Link}

Table 2: Mapping from MiniMD to LaTeX

```
\documentclass{article}
\usepackage[colorlinks=true]{hyperref}
\begin{document}
% Beginning of MiniMD code
% End of MiniMD code
\end{document}
```

Figure 4: LaTeX template