Adapting a Markdown Compiler's Parser for Syntax Highlighting

Ali Rantakari

June 3, 2011

Abstract

Markdown is a presentational markup language that is designed for readability. It is also very context sensitive, which makes it difficult to highlight correctly using the syntax highlighting mechanisms available in existing text editor programs. In this report we discuss the implementation of a Markdown syntax highlighter that is able to handle this context sensitivity by adapting the parser of an existing Markdown compiler. We evaluate five compilers for qualities we would like to see in our highlighter (correctness, embeddability, portability, efficiency), as well as for how easy they are to modify, and pick *peg-markdown* as the one to take the parser from. We then adapt its parser for syntax highlighting purposes and integrate it with the Cocoa GUI application framework. The end result is a portable syntax highlighting parser for Markdown, written in C, as well as Objective-C classes for using it in Cocoa applications.

Contents

1 Introduction

Markup languages provide ways to apply metadata to segments in a body of text by using a specific syntax (which is defined by the language grammar) so that the metadata description becomes part of the body of text itself. As an example of what these syntaxes might look like, consider the following two lines, showing an application of regular and strong emphasis to sections of text using the HTML and LaTeX markup languages:

```
In HTML, we use <em>the em and strong tags</em> to add
<strong>emphasis</strong>.
```

```
In LaTeX, we use \emph{the emph and bf commands} for the same {\bf effect}.
```

While the metadata supported by a markup language may generally describe almost anything, in this assignment we concentrate on *presentational* markup: metadata that describes how the text should be formatted for presentation.

HTML is the standard format for presentational markup on the World Wide Web; web pages are written in it and interpreted by web browsers in order to present them to the end users. LaTeX is another presentational markup language, but its primary use is in preparing scientific documents in academia, where the document is often translated into a PDF format and then rendered for presentation to end users by a PDF reader application. Both of these languages work well for their respective purposes but because readability by humans has not been an important factor in their design, they are often cumbersome to read as-is, requiring this translation before presentation to readers.

Markdown, on the other hand, is a markup language with the explicit goal of being as readable as possible. Markdown documents can be transformed to other formats (such as HTML or LaTeX) for presentation to readers, but they should also be presentable even in their raw form. [8] As a quick demonstration of this quality, consider the following Markdown-formatted example, which is equivalent to the previously shown HTML and LaTeX examples:

```
In Markdown, we use _underscores or asterisks_ to add **emphasis**.
```

Although this example is quite simple, we can already see the difference in readability. Markdown supports the same kinds of basic document formatting features as HTML and LaTeX but strives to keep the syntactic notation as light as possible.

Syntax highlighting is the process of recognizing and highlighting — usually with colors but sometimes with typographic styles as well — syntactic elements in a textual representation of a language. It is useful for both readers and writers by making it easier to perceive the structure of the text in terms of the used language: for example an HTML syntax highlighter might display tag notation like `<em>` and `</em>` in a specific color but use the default text color for content in between (just like in the examples previously shown in this report, where the syntactic elements are printed in magenta and the rest of the text in black).

Although syntax highlighting is most useful in cases where it might be difficult to differentiate actual content from the markup, there are good reasons why it also

helps writers using Markdown, even though its syntax is designed to "stay out of the way" of actual content as much as possible. One such reason is the fact that Markdown is very *context sensitive*: the interpretation of some part of a Markdown-formatted document might depend completely on the content surrounding it. It is thus useful for writers to have a syntax highlighter inform them how their text will be interpreted by a compiler. This also makes Markdown difficult to highlight with the syntax highlighting mechanisms available in common text editor programs. We discuss Markdown's context sensitivity (and the implications it has on the difficulty of highlighting it) more in section 2.2.

The goal of this assignment is to produce a Markdown syntax highlighter that is able to handle the complexities of the language while being fast to execute and easy to take into use in GUI applications. In order to obtain a solution to the first requirement for free, as it were, we will utilize the parser of an existing Markdown compiler instead of writing a new one from scratch.

In the following section we will discuss the language more in depth, and particularly what complexities therein our highlighter must be able to handle. In section 3 we will outline the requirements for this assignment, and in section 4 evaluate existing compilers and pick the one to take the parser from. Section 5 describes the implementation of the syntax highlighter and in section 6 we will evaluate it against our requirements. Section 7 discusses ideas for possible future work and the last section, 8, sums up the report.

2 The Markdown Language

Markdown, designed by John Gruber, is a markup language with the primary goal of being as readable as possible.[1] The syntactic elements it defines map directly to HTML, which is the most common format Markdown documents are translated to (using compilers built for the purpose). It is used on popular websites such as *StackOverflow*, *GitHub* and *Reddit* for formatting user-submitted content, and some content management systems support it as an input format. [8, 23]

We briefly compared the readability of Markdown, HTML and LaTeX in the introduction, but in order to understand the difference a little better we'll consider the following simple document: a second-level header, followed by an unordered (bulleted) list. The list has two items, the first of which contains two separate paragraphs, and the second only one. The text in the first item's paragraphs has strong emphasis for the first words and a section of the text in the second list item is to be displayed as "code" (or "verbatim").

This is the HTML representation:

[1] The author uses the same name to refer to his Markdown to HTML compiler, but in the name of clarity in this report we will use the name *Markdown.pl* to refer to the compiler, and reserve *Markdown* for the language itself.

```
<h2>Section title</h2>
<ul>
  <li>
    <p><strong>First</strong> paragraph in first item</p>
    <p><strong>Second</strong> paragraph in first item</p>
  </li>
  <li><p>Second item with <code>inline code</code></p></li>
</ul>
```

We can see how the metadata syntax in HTML takes a lot of space in the text, making it difficult to read. The following is the LaTeX version of the same content:

```
\subsection{Section title}
\begin{itemize}
  \item {\bf First} paragraph in first item

        {\bf Second} paragraph in first item
  \item Second item with $inline code$
\end{itemize}
```

This is not quite as verbose as the HTML version but still not very readable. Contrast these examples with the Markdown version:

```
Section title
-------------
- **First** paragraph in first item

  **Second** paragraph in first item

- Second item with `inline code`
```

The difference is striking; the Markdown formatting mostly stays out of the way and allows the reader to consider the structure of the actual content without spending a lot of time trying to separate it from markup.

In the following subsections we will briefly explain some prominent features of Markdown as well as discuss the main problems in highlighting its syntax correctly.

2.1 Prominent Features of Markdown

All of the syntactical elements that Markdown defines have an equivalent element in HTML, but the opposite is not true — Markdown only supports a small part of what can be done with HTML. Another fundamental aspect that Markdown has in common with HTML is the separation of *block-level* and *span-level* elements. Paragraphs, lists and headers are examples of the former and links, emphasis and code of the latter. [8]

Paragraphs in Markdown are separated from each other by blank lines, and may not be indented. An indented paragraph becomes a "code block" to be presented verbatim:

```
This is a paragraph.

This is not the same paragraph as the one above because there is an
empty line in between. The indentation in the following paragraph
makes it a code block:

    This is a code block.
```

Lines beginning with the hash symbol are headers. The number of hashes specifies which level the header is:

```
# First-level header
## Second-level header
### Third-level header
```

First and second-level headers have an alternative syntax where the line with the header text is "underlined" by equal signs or hyphens on the following line:

```
First-level header
==================

Second-level header
-------------------
```

Unordered lists can be created by using hyphens, pluses or asterisks to begin lines with items of the list. List items may contain multiple paragraphs, and sub-lists may be created by indenting:

```
- First item

  Second paragraph in first item

- Second item
    - Item 2.1
    - Item 2.2
- Third item
```

Ordered lists work similarly, with the exception that items are specified by beginning lines with a number and a period:

```
1. Uno
2. Dos
3. Tres
```

Emphasis is added by surrounding a segment of text with underscores or asterisks. Using two of whichever symbol makes the emphasis strong:

```
Here we create _normal emphasis_ and __strong emphasis__ using underscores.
We can also use *asterisks* for the **same effect**.
```

In addition to code blocks, verbatim presentation can be specified with inline code spans, which are created by surrounding text with backtick quotes:

```
The method signature ‘destroy(Planet *target)‘ is to be presented verbatim.
```

This is a very cursory overview of what is available in the Markdown language, but it should be sufficient for understanding the rest of this report.

2.2 Why Markdown Syntax is Difficult to Highlight

Syntax highlighting is in common use for programming, where it is very important to be able to easily structure and comprehend the program code. Since the number of programming languages in use today is quite large, most implementations of syntax highlighting in programming editors focus on making it easy to add definitions for new languages, at the expense of the expressive power of the syntax definition system — in most cases this means using regular expressions to match elements in the language. This makes them inadequate for highlighting Markdown because it is much more difficult to highlight correctly than most programming languages.

As an example, consider the following Markdown-formatted text:

```
Here is some code:

    NSString *s = @"hello world";
    if ([s hasPrefix:@"hello"])
        NSLog(@"greeting detected");
```

The first line is a paragraph, then there is an empty line to separate this paragraph from the following lines, and the next three lines define a code block. We know it is a code block because they are defined by indenting the lines with four spaces or one tab. This document, transformed into HTML, would look like this:

```
<p>Here is some code:</p>

<pre><code>NSString *s = @"hello world";
if ([s hasPrefix:@"hello"])
    NSLog(@"greeting detected");
</code></pre>
```

But if you were to add a hyphen and a space to the beginning of the first line, the last three lines wouldn't define a code block anymore:

```
- Here is some code:

    NSString *s = @"hello world";
    if ([s hasPrefix:@"hello"])
        NSLog(@"greeting detected");
```

The above would translate to the following HTML:

```
<ul>
<li><p>Here is some code:</p>

<p>NSString *s = @"hello world";
if ([s hasPrefix:@"hello"])
    NSLog(@"greeting detected");</p></li>
</ul>
```

The added hyphen made the first line a list item definition instead of a paragraph, and because of that any following lines indented with four spaces or one tab will continue the contents of that list item instead of defining a code block. If you wanted to keep the indented lines as a code block (inside the list item, however) you would have to indent them a second time. This shows how you cannot simply assume that every indented block of lines defines a code block; whether it does or not depends on its context, and this context can be of arbitrary length — checking whether the previous block defines a list item is not enough since such a definition may exist however early in the text and the lines in between define other content for it. This example illustrates just one particular aspect of Markdown's *context sensitivity*; there are many more.

Many programming languages contain constructs that essentially make them context sensitive as well, such as identifiers having to be defined before their use, or a function call having to specify the same number and types of arguments than in the definition of the function. [1, pp. 215–216] Syntax highlighters for such languages usually ignore these issues, leaving them to the later semantic analysis phase in the compiler. Ignoring context sensitivity in this way makes more sense for programming languages than it does for Markdown because for programmers the process of writing instructions (that is, code) and having the computer interpret them are, to an extent, conceptually separate. The lexical structure of program code is usually a good representation of the programmer's intent, and thus highlighting it helps the process of writing it — some context-dependent syntactic errors may be highlighted in the editor only after compiling, or not at all. Since Markdown is a markup language, the intent of the writer and the computer's interpretation of it are conjoined to a much higher degree.

Another difference between highlighting a programming language and Markdown is that the latter is *not well defined*, while the former usually is. This is important because it greatly simplifies the highlighting of programming languages with simple lexical analysis and parsing, and ensures that the number of corner cases where more complex mechanisms would need to be used to highlight a certain input correctly are much more rare. In contrast, it is much more common for writers using Markdown to adhere to the syntax definition but end up with input that is interpreted in a way they did not intend.

3 Project Requirements

In this section we will specify the functional and non-functional requirements for the syntax highlighter implementation. These requirements are used in the following

chapters as the basis for evaluating both existing compilers as well as our finished implementation.

For the purposes of defining these requirements we separate the implementation into two distinct parts: the *parser* and the *highlighter*, and list the requirements for them individually.[2] We want to be able to use the parser on many different platforms and with many different GUI frameworks, but the highlighter we implement using Apple's Cocoa framework.

3.1 Functional Requirements

- *Parser:* Given the UTF-8 -encoded contents of a Markdown document, produce language element occurrences, each with type and position information.
- *Highlighter:*
  - Given a GUI text editor widget, feed the text it contains to the parser,
  - Highlight the text in the GUI widget based on the parser's output,
  - Provide option to automatically parse and highlight the text contained in the widget whenever it changes,
  - Provide configuration interface for element highlighting order (for example, links before emphasized text) and styles (colors, fonts etc.) used for different elements.

3.2 Non-functional Requirements

- *Correct:* The parser should recognize language elements as correctly as possible, and in particular we want it to significantly improve upon what is possible with the regular expression -based highlighting mechanisms in existing editors. In other words, we want the parser to be on the same level of correctness as existing Markdown compilers. This requirement is considered fulfilled if the parser does not regress from the behavior of the compiler it was adapted from.
- *Embeddable:* Both the parser and the highlighter should be able to be taken into use in (that is, embedded into) existing and new programs as easily as possible. Embedding the parser into two separate proof-of-concept applications that use different GUI frameworks, and the highlighter (with the parser) into two separate Cocoa applications fulfills this requirement.
- *Efficient:* The parser and highlighter should be reasonably fast and the parser should not use excessive amounts of memory. As a rough guideline, parsing and highlighting a 50 KB document in less than half a second while consuming less than 10 MB of memory fulfills this requirement.

[2]This separation is done for the requirements definition only; in the rest of this report the term "highlighter" will refer to both of these parts together.

- *Portable:* The parser should be portable to the most common desktop operating systems.[3] Compiling a functional parser on Linux, OS X and Windows systems fulfills this requirement.

4 Evaluation of Existing Parsers

Compilers that are used for transforming a document from one format to another generally have two distinct parts: a *front-end* that interprets the input and transforms it into some kind of an intermediate form, and a *back-end* that generates the output based on this intermediate representation (programming language compilers — as opposed to Markdown compilers — may also include multiple optimization phases). The front-end does its job by performing lexical analysis, or *scanning*, and syntactic analysis, or *parsing*. The job performed by the back-end may also be called *code generation*. Scanning and parsing can be performed during the same pass through the input instead of in completely separate steps, so for the sake of brevity in this report we will use the term "parser" to refer to the whole front-end, including the lexical analyzer. [1, pp. 4–8]

In this section we will evaluate some parsers in existing Markdown compilers. We will take the quality attributes from our non-functional requirements as the basis for this evaluation, the idea being that if the parser exhibits these qualities, then the syntax highlighter based on it would exhibit them as well. In addition we also want to consider *modifiability* because we certainly want to be able to easily extract the parser from the compiler and modify it for our purposes.

Appendix 9.1 contains version information for all relevant tools used for this evaluation, as well as for the tested compilers.

4.1 Selection of Compilers to Evaluate

There are many implementations of Markdown compilers, so we want to trim down the list to the most promising ones (in terms of our desired quality attributes) before we start the evaluation.

Because we want the highlighter to be easy to embed into existing and new programs, we try to avoid compilers written in dynamic, high-level programming languages such as Perl, PHP, Python or Ruby — embedding a parser written in one of these languages would entail embedding a whole new runtime environment as well (unless, of course, the host application happens to already run in that environment). The same reason also leads us to rule out implementations in the Java language. This leaves us *Discount* by David Parsons and *peg-markdown* by John MacFarlane (which are written in C) as well as *Cpp-Markdown* by Chad Nelson (which is written in C++).[4] As an exception to the first rule, we include the Lua implementations *Lunamark* (also by John McFarlane) and *Markdown.lua* (by Niklas Frykholm) due

[3] The highlighter need not be portable as is; it is best to custom-tailor a corresponding component for each application (or application framework) where the parser is used.

[4] *libupskirt* by Natacha Porté is another C implementation but the author of this report was not aware of it at the time, which is why it is not included in this evaluation.

to the reputation of the Lua language for being very fast as well as easy and light to embed. [12]

Although *Markdown.pl*, the original implementation by John Gruber, is written in Perl (and as such not considered as a basis for the highlighter implementation), we will include it in the "correctness", "efficiency" and "modifiability" parts of the evaluation as a baseline to provide context for the other implementations.

4.2 Embeddability and Portability

Although we have earlier considered these two qualities to be distinct from each other, it is useful to discuss them together here, since many things in the parsers that make them embeddable also make them portable, and vice versa. These attributes were already a key factor in our selection criteria but here we look at the selected compilers a bit more closely. The main indicators we consider are the programming language (or a variant thereof) used and the possible existence of third party dependencies (including *their* embeddability and portability).

Discount is written in C, and the author notes that for him, it builds on SLS Linux, MacOS 10.5, FreeBSD 4.8, and RHEL3, but promises no Windows support. It has no third-party dependencies. [17]

Peg-markdown is written in *"portable ANSI C"*. It also depends on the *glib2* library (which is available on *nix platforms but needs to be cross-compiled with MinGW for Windows), but based on the source code it is apparent that this dependency is used mainly for the code generation, and not nearly as much in the parser. [13]

CppMarkdown is written in C++ and depends on some parts of the *Boost* library (which is supported on all of our target platforms) — a few header-only libraries and the compiled regular expression library. The author says that it was designed to be integrated into C++ programs and that it is operating system -independent. [16]

Lunamark is written in Lua and uses the *lpeg* library. The author describes it as *"very portable since both Lua and the lpeg library are written in ANSI C"*. [14] Indeed, the Lua web site describes it as highly portable (it builds on all platforms that have an ANSI/ISO C compiler) and easily embeddable (into programs written in a multitude of different languages, such as C, C++, Java, C#, Python and Ruby). [12] Since Markdown.lua is written in the same language but has no dependencies, the same applies to it as well.

4.3 Correctness

It is important for the syntax highlighter to be correct in what it highlights, but evaluating this is difficult because the concept is not clearly defined. The closest things to an accepted standard for the Markdown language are the syntax description document[5] and the behavior of the original compiler implementation (Markdown.pl). Unfortunately the syntax description is ambiguous on many points and there are corner cases where people disagree with the decisions made by the original compiler. [13]

[5] Available at `http://daringfireball.net/projects/markdown/syntax`

John Gruber provides *MarkdownTest*, a simple test suite for Markdown to HTML compilers, which tests most of the language's features with simple test inputs, making sure that the HTML output is as expected. [9] This suite does not really test any difficult edge cases. Another available test suite, one that does try to test some edge cases, is *MDTest* by Michel Fortin, the author of PHP Markdown. [7] MDTest contains three different test suites, only one of which ("Markdown") tests standard language syntax compliance; the other two ("PHP Markdown" and "PHP Markdown Extra") test compliance with custom language extensions. Only the first one is useful to us since we are not interested in extensions to the language.

We run both tests on all six compilers. For Discount, we use the `-f 0x4` argument to make it not do Smartypants-style substitutions because the tests don't expect them to happen. Table 1 details the results, with the percentages rounded to the closest integers.

Table 1: MarkdownTest and MDTest results (percentage of tests passed)

Compiler	MarkdownTest	MDTest (Markdown suite)
Discount	100 %	82 %
peg-markdown	95 %	74 %
Cpp-Markdown	100 %	83 %
Lunamark	73 %	60 %
Markdown.lua	77 %	87 %
Markdown.pl	95 %	91 %

Discount passes the full MarkdownTest but some problems that MDTest reveals in it are related to it generating invalid HTML in some edge cases as well as its handling of parentheses in URLs and empty HTML paragraph tags.

Peg-markdown passes all but one of the tests in MarkdownTest, the correctness of which the author argues against, saying that it is better this way and that it is still consistent with the Markdown syntax definition. [13] From looking at peg-markdown's failed MDTest results, it seems that the same reasoning would apply there for some of the failings, but other issues that it has there include escaping of characters as well as parentheses in URLs.

Cpp-Markdown, like Discount, passes MarkdownTest fully. In MDTest it has trouble with escapes, inline HTML and parentheses in URLs.

Both Markdown.lua and Lunamark fail to recognize certain variants of the link syntax. Markdown.lua also fails to correctly handle edge cases related to inline HTML, escapes and URLs with parentheses. Lunamark has additional problems with whitespace and image definitions.

Markdown.pl fails in some parts of MDTest simply because it adds empty `title` attributes to link elements where none was specified, and in one part of MarkdownTest because it indents the output wrong for code blocks within blockquotes.

Many of the cases where these implementations fail seem to only be due to code generation, and as such not be relevant for syntax highlighting (which would only

utilize the parser), but every single compiler (with the exception of Markdown.pl) also has small problems that would affect syntax highlighting.

4.4 Efficiency

We separate efficiency into two distinct concerns: execution time and peak memory usage.

One problem in evaluating the efficiency of the parsers in the studied compilers is that although it would be ideal to separate them from the rest of the compiler code and benchmark them in isolation, this would require a prohibitive amount work, or may even be completely unfeasible (Markdown.pl in particular is not very modular in terms of separation between parsing and code generation). Instead we benchmark the compilers as a whole, with the assumption (or hope) that the portion of the execution times taken by output generation would be fairly constant between all six, and that the relative magnitudes of peak memory usages would be sufficient in producing a meaningful comparison.

Another issue might be the use of non-optimal C or C++ compiler settings for some of the evaluated Markdown compilers — it would be best if optimal settings were used in each case. Instead of somehow determining what these optimal settings are we simply trust that the authors have selected good defaults. Thus we use the default configuration, set in their respective build settings, for Discount, peg-markdown and Cpp-Markdown. For all three we use the same GCC compiler.

We use two Markdown files for these benchmarks: a file with "normal" formatting (that is, formatting that tries to adhere to the syntax definition) and another with more eccentric formatting (that is, formatting that sometimes strays from the syntax definition). These are called *normal.md* and *eccentric.md*, respectively. The file contents are detailed in the Appendix sections 9.2 and 9.3.

4.4.1 Execution Time

In order to measure the execution times we modify each compiler to calculate the time interval between start of parsing and end of code generation, using the `gettimeofday()` C function in peg-markdown, Cpp-Markdown and Discount, `gettimeofday()` from the `Time::HiRes` Perl module in Markdown.pl and `socket.gettime()` from the LuaSocket library in Lunamark and Markdown.lua. We then make the compilers output this interval value instead of the generated HTML.

For the first test we multiply the contents of *normal.md* 20 times and the contents of *eccentric.md* 130 times, ending up with total file sizes of 139 KB and 163 KB. We compile these files 100 times with each compiler (for the Lua implementations, both with the standard Lua runtime and with LuaJIT[6]) and calculate averages for the results — these are shown in table 2. The times are rounded to two decimals.

Discount is clearly the fastest, with peg-markdown trailing quite close behind. Markdown.pl, Lunamark and Markdown.lua are slower than the fastest two by more than an order of magnitude, and Cpp-Markdown is in between these two groups. As

[6]LuaJIT is a just-in-time compiler that can act as a stand-in for the standard interpreter. See `http://luajit.org` for more information.

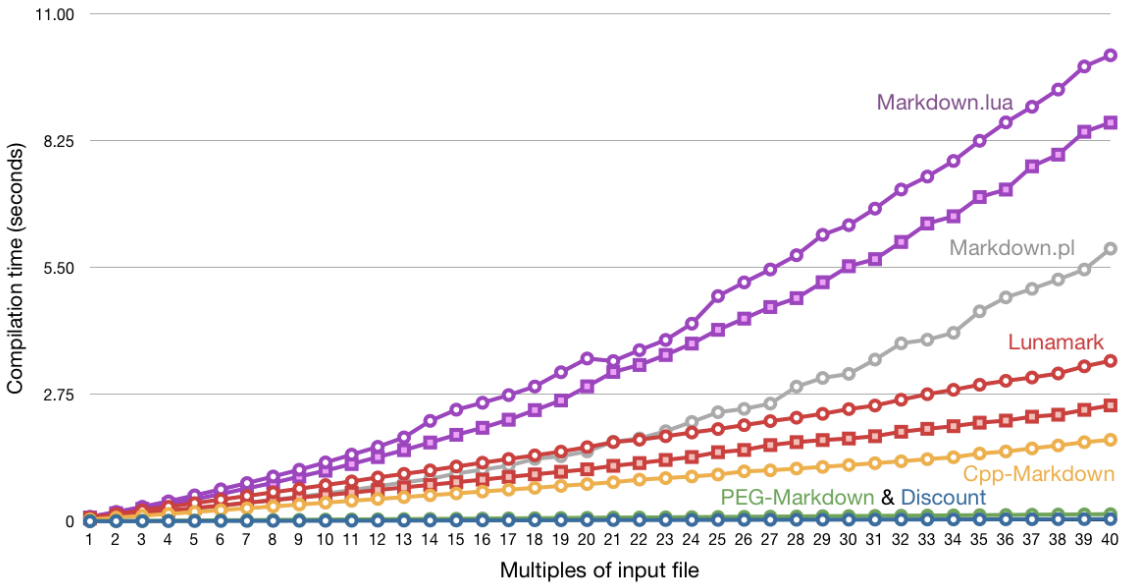

Figure 1: Growth of compilation times for longer inputs (square, filled data points denote LuaJIT results)

an interesting anomaly there was something in the "eccentric" input file that made Markdown.pl take half a minute to get through it. Also, Markdown.lua was curiously the only one that did not take significantly longer to process that file, but did it in fact faster than "normal.md" (with LuaJIT, noticeably faster).

In addition to measuring average compilation times for large inputs, we also measure the increase in compilation time as the size of the input increases. We concatenate the contents of "normal.md" into 40 different files, each with a higher multiple of its contents, and run each compiler for all of these files (ten times per file, averaging the results). The results have been plotted into figure 1.

Table 2: Compiler comparison: average compilation times (in seconds)

Compiler	normal.md (x 20)	eccentric.md (x 130)
Discount	0.02	0.04
peg-markdown	0.08	0.13
Cpp-Markdown	0.81	1.54
Lunamark	1.62	4.17
Lunamark (LuaJIT)	1.13	2.82
Markdown.lua	3.32	3.22
Markdown.lua (LuaJIT)	2.84	1.18
Markdown.pl	1.56	27.01

Almost all of the compilers seem to exhibit nearly linear time complexity in terms of the length of the input. The only ones for which a slight upwards curve is present are Markdown.lua and Markdown.pl. These curves might imply superlinear complexity.

4.4.2 Memory Usage

We use the Valgrind Massif tool (with the arguments `--depth=1` and `--trace-children=yes`) for measuring the peak heap memory usage of the four compilers. Table 3 displays these results.

Table 3: Compiler comparison: peak heap memory usage (in MB)

Compiler	normal.md (x 20)	eccentric.md (x 130)
Discount	0.65	1.90
peg-markdown	4.38	6.70
Cpp-Markdown	1.24	3.94
Lunamark	5.77	10.87
Markdown.lua	2.71	9.61
Markdown.pl	6.19	9.36

Again, Discount stands out from the group as the one needing clearly the least amount of memory, with Cpp-Markdown coming in second. peg-markdown uses less of it than Lunamark and Markdown.pl but this time it is much closer to these two than it is to Discount. Memory usage tripled for "eccentric.md" in comparison to "normal.md" with Discount, Cpp-Markdown and Markdown.lua.

All in all the memory usage feels appropriate; none of these seem to use an excessive amount of it. When code generation is removed and just the parser is used, memory consumption is likely to noticeably decrease.

4.5 Modifiability

All of the quality attributes discussed above have to do with the desired end result, the idea being that the highlighting parser would inherit them from whichever parser it is based on. This one, however, is about the feasibility of the project in general — how easy it would be to modify a parser for the purposes of syntax highlighting — and is thus perhaps the most important one.

Although not an end in and of itself, the length as well as the comment density of the source code of a compiler can be an indicator of how easy it would be to modify. We use the *Ohcount* software to calculate SLOC counts for each of the considered compilers. Table 4 displays these results.

Discount has fared well in terms of all of the other quality concerns, but unfortunately it is by far the longest program in terms of source code, and it seems very difficult to separate the parser from it and modify it for syntax highlighting purposes. There is no declarative element to the parser (that is, no domain-specific language is

used to define the parser behavior); it is instead hardcoded in a relatively low-level language (C), and the code assumes throughout that the parsing is done in order to transform the input into another format, instead of to simply interpret it.

The author mentions as much in his message to the *markdown-discuss* mailing list: [18]

> *In regards to the actual question, I do have code in discount that creates a block-level parse tree, but I don't build a parse tree for the content inside paragraphs. And I don't keep track of byte offsets into the source. It wouldn't be difficult to add the offset to the start of each paragraph into the data structures; you'd need to keep track of byte & line offsets when lineilating [sic] the input into the library, and then each paragraph object could easily get to that information for its first line.*
>
> *If you wanted to do the same for span-level markup, you'd be completely on your own. I'm sorry about that, but I never considered using discount as an interpreter instead of a compiler, and it would probably take a fairly extensive redesign to get it to work nicely without suffering a staggering performance penalty.*

Peg-markdown and Lunamark, on the other hand, define a Parsing Expression Grammar that describes the Markdown language, and then perform the parsing based on that. Peg-markdown defines the grammar with a specialized domain-specific language similar to *Lex/Yacc* syntax, and then C code for the parser is generated from that description by using the *peg/leg* parser generator. Lunamark, in contrast, describes the grammar in Lua syntax, using constructs provided by the *lpeg* pattern-matching library – so everything happens at run-time. Both of these seem very easy to modify.

Cpp-Markdown and Markdown.lua both implement hardcoded parsers. The separation between parsing and code generation seems more explicit in Cpp-Markdown but it has more code and a lower comment density. Subjectively, the code in Markdown.lua also seems a little more understandable.

Table 4: Compiler comparison: non-comment source lines of code (NCSLOC), percentage of comments from all source lines of code

Compiler	NCSLOC	Comments
Discount	4475	13.7 %
peg-markdown[7]	1813	9.3 %
Cpp-Markdown	1713	7.9 %
Lunamark	359	0.6 %
Markdown.lua	959	20.4 %
Markdown.pl	927	31.0 %

[7]The SLOC count for peg-markdown includes the PEG grammar file; the count for code only is 1232, of which comments 11.1 %.

4.6 Choice of Parser

Since execution speed is something that is bound to be inherently based on the overall design — and thus not something we can easily significantly improve — we lean towards the fastest of the tested compilers. This excludes the Lua implementations (which also used more memory and did not fare well in the language test suites). Discount, the undisputed speed victor, is also out due to its design not lending itself to what we want to do. Since peg-markdown and Cpp-Markdown were close in terms of correctness, portability and embeddability, but the former noticeably faster and significantly easier to modify, we choose it as the basis for our highlighting parser. Peg-markdown's higher memory consumption is a concern but we trust it will be significantly reduced as a result of removing code generation.

5 Implementation

In this section we will discuss the implementation of the syntax highlighter. This consists of two major parts: adapting the peg-markdown parser for syntax highlighting and integrating the result into a Cocoa GUI application.

5.1 Adapting the Compiler's Parser for Syntax Highlighting

There are fundamental differences in how compiler parsers and our syntax highlighter parser want to process the input text. The compiler needs to transform the content from one format to another, so it is interested in retaining the *abstract structure* of it as well as the "actual" contents (that is, contents that are not used solely for representing structure).

A common, generic way of representing hierarchical structure is by constructing an abstract syntax tree (AST) of the input document where the types and contents of language elements are stored in each node. [1, pp. 69–70] This is what peg-markdown does, and what the parsers of other Markdown compilers need to do as well.[8] The data structure needs to be a tree because Markdown, like many programming languages, contains elements that may be nested inside other elements, and this nesting structure needs to be retained when the input is transformed into HTML or any other format that represents structure by having some elements act as containers for others.

Consider the following Markdown-formatted text:

```
1 ## Hello world
2
3 Lorem _ipsum dolor_ sit amet.
```

The AST representing the above input could look like something like this:

[8] Even if the compiler does not explicitly construct an AST, it will still have to perform the parsing in a manner that recognizes the hierarchical abstract structure of the input document.

```
+-- HEADER 2 "Hello world"
+-+ PARAGRAPH
  +-- TEXT "Lorem"
  +-- EMPHASIZED "ipsum dolor"
  +-- TEXT "sit amet."
```

Notice how characters that mark language elements (like the hashes that mark the beginning of a header or the underscores that delimit emphasized text) are omitted in the AST data and only the "actual" contents are stored. This is one reason why this kind of a representation does not make sense for a syntax highlighter: we need to find the character offsets in the text where language elements occur, including the characters that define or delimit them; we are not necessarily interested in separating "actual" content from other characters in the input text. We also have no use for an internal representation of the structure of the input document (insofar as it does not affect correct recognition of elements in the input), tree or otherwise.

5.1.1 Basic Modifications

The *peg/leg* parser generator's grammar syntax recognizes *actions* — blocks of C code that can be attached to productions in the grammar. When a production is matched by the parser, its corresponding action will be executed. This is essentially an application of *syntax-directed translation*. [1, ch.5] Peg-markdown constructs the AST by creating internal representations of language elements in these grammar actions (and appending them to the tree), and in our highlighter we want to similarly store information about matched language elements, in particular their type and the start and end offsets of their occurrence in the input text.

Peg/leg makes certain information available to grammar actions, including the input text that was matched in the production (as delimited by a specific syntax in the production definition), but this information does not include start or end offsets. Fortunately *greg*, a modified version of peg/leg, adds support for accessing the start and end offsets in actions. Greg improves upon peg/leg in other ways as well — most notably by generating parsers that are re-entrant, which means that they can safely be used from multiple threads. [22]

Instead of producing a tree structure our highlighter stores language element occurrences in an indexed array of linked lists. We want to be able to choose the order in which different language elements get highlighted so we index the array by element type in order to have random access to the linked lists of each type. We use linked lists for the element occurrences because they are fast to manipulate and because we don't need random access into them.

5.1.2 Handling Post-processing Parser Runs

The biggest complicating factor in working with peg-markdown's parser is the fact that it is not able to parse all Markdown documents in a single pass. Instead, for some difficult container elements — namely blockquotes and lists — it creates "raw" elements into branches in the syntax tree, containing the part of the input that it recognizes as representing the contents of the container element but is not able to

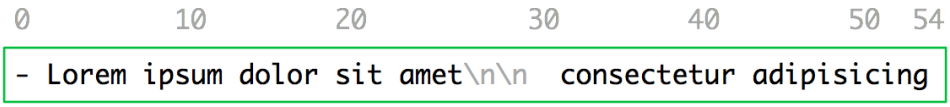

Figure 2: Example original input

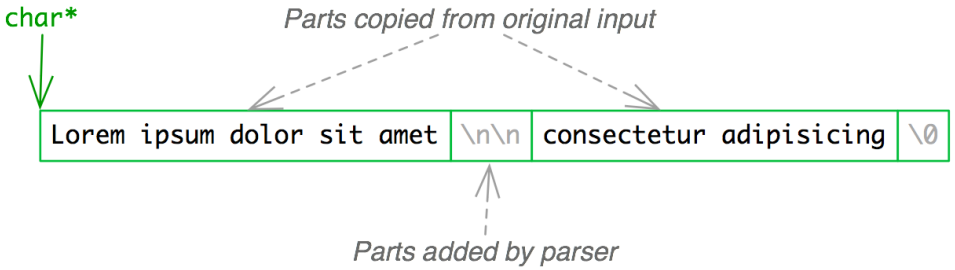

Figure 3: "Raw" string created by peg-markdown for further parsing runs (corresponding to original input in figure 2)

correctly parse. Then as a second parsing step it iteratively runs the parser on the strings contained in each of these raw elements, replacing them in the tree with their corresponding parsing results. The problem this brings is the fact that the offsets of matched elements in these post-processing parser runs won't match the locations in the original input.

One way to make the offsets match would be to store the start offsets of each "raw" element and then adjust the offsets of elements created in the post-processing parsing runs by adding this value to them, but it becomes obvious this won't work when you notice that the text added to "raw" elements by the parser is not a contiguous span from the original input, but instead a new string that the parser constructs from select parts of the original input, in some cases even adding additional strings in between. So we need the parser to be able to step through and parse lists of arbitrary disjoint segments in the original input and map the "virtual" offsets within this list of segments to the corresponding real offsets in the original input. In addition, we need to be able to insert arbitrary strings into this list for the parser to read but not take into account when determining the mapping of virtual to real offsets.

We implement this by modifying the parser to traverse the input string not linearly from start to end but instead based on indexes stored in a linked list of span markers (each marking a start and end offset for a section in the original input that the parser should consume). Included in this list may also be elements containing additional strings that the parser should consume ("extra text" elements). Whenever language elements are found during parsing, their offsets are adjusted to match the original input by linearly walking through this list of offset span markers (ignoring "extra text" elements) and finding the ones where the virtual start and end offsets of the language element fall. During this process we must also take into account the fact that the list of spans may be disjoint — if a match begins in one span and ends in another we must split it into parts in order to exclude segments between the spans.

Consider the example string presented in figure 2. If we take this to represent a substring in some larger original input that the parser needs to handle in post-

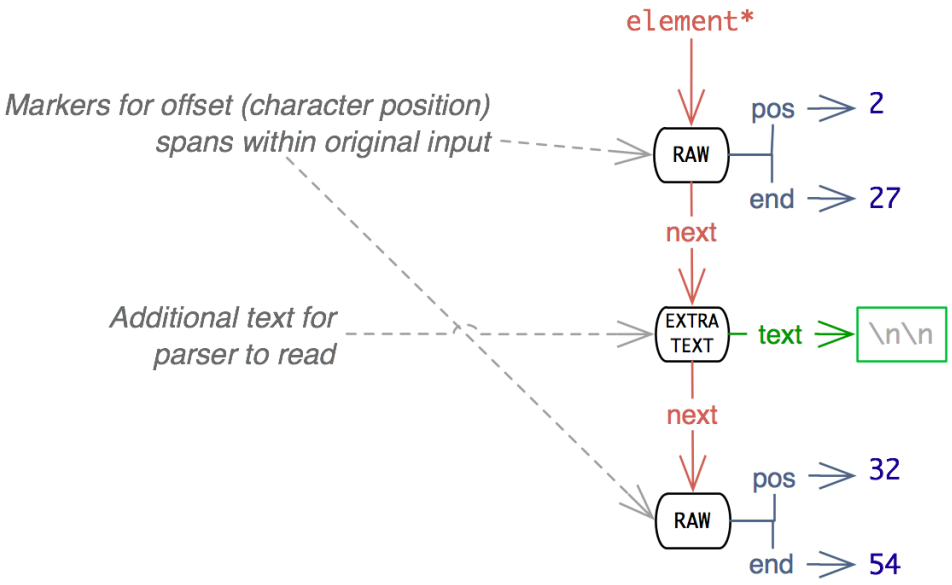

Figure 4: Parsing data structure created by our highlighter parser for further parsing runs (corresponding to original input in figure 2)

processing, and figure 3 the "raw" string that the peg-markdown compiler would create based on it, then figure 4 is the corresponding data structure in our highlighting parser, defining the segments to parse in the original input as well as an additional string in the middle to include in the parsing but not in the offset calculations.

5.1.3 Supporting UTF-8 String Encoding

UTF-8 is one of the many encodings for Unicode character strings. It is a variable width encoding — that is, a character in UTF-8 may be encoded in up to four bytes instead of always just one, like in ASCII. It is compatible with ASCII in that every valid ASCII encoding is a valid UTF-8 encoding. [20]

The original peg-markdown compiler handles characters in input strings in a per-byte manner due to the use of the C `char*` type to represent strings in the peg/leg (and greg) parser generator. The compiler does, however, support UTF-8 input by treating all multibyte (non-ASCII) characters as alphanumeric (in the few cases where such a distinction needs to be made). [15] Due to being reliant on character offsets our highlighting parser needs random access into the original input buffer, and UTF-8, being a variable-width encoding, does not provide that.

UTF-32 is a constant-width encoding so using it with the `wchar_t*` type would give us random access, but this would entail converting the whole input string (work we would like to avoid doing) and increasing memory consumption a little. More importantly, we would also have to make fundamental changes into peg/leg (or greg) to make it use and support wide character strings. Since the parser only cares about ASCII characters (and considers all non-ASCII bytes alphanumerics) we instead simply strip the continuation bytes (that is, the bytes after the first one in a sequence that defines a single non-ASCII Unicode code point) from the input before processing it. This is easy because in UTF-8 the two most significant bits of all continuation

bytes (and, imporantly, of *only* continuation bytes) are `10`. [19, pp. 94] [20]

A unicode *byte-order mark* (BOM) is a metacharacter[9] in the beginning of a document that describes the byte order used. Even though the BOM does not indicate byte order in the beginning of a UTF-8 data stream (the byte order is always the same in UTF-8), it may be present as an indication that the otherwise unmarked data is encoded using UTF-8. [19, 21]

With the changes described above our parser can support UTF-8 -encoded input, but still the existence of a BOM in the beginning can cause all of the indexes for parsing results to be incorrect. If the BOM character is being interpreted by the parser as a normal character in the input but as a non-displayable metacharacter by the program displaying the highlighting results, the highlighted positions in the displayed document will be all off by one character. To remedy this in the parser we simply check for the existence of a BOM in the beginning of an input and discard it if found.

5.2 Integration Into a GUI Application

Cocoa is an Objective-C GUI application development framework in Apple's Mac OS X operating system. In order to integrate the parser into Cocoa GUI applications we implement highlighting functionality that can be connected to a text editor widget (represented by the `NSTextView` class) and do the things specified in our functional requirements (in section 3.1).

One way to connect our desired highlighting functionality to instances of `NSTextView` would be to subclass it, but this would prevent integration into applications that already have subclassed it (the Objective-C programming language does not support multiple inheritance). Another way would be to implement a *category* (a method of attaching additional functionality to Objective-C classes without subclassing them) for `NSTextView` but categories cannot implement persistent state (that is, instance variables) which we would need. [10] We thus create a separate class that can hold a reference to its target `NSTextView`.

Our highlighter class uses the Cocoa notifications mechanism to receive notifications from the text editor widget whenever the text it contains changes. When such a change occurs, the highlighter runs the parser on the text. After the parser is done, the highlighter goes through the results and applies styles to the corresponding sections of text in the `NSTextView` by using methods in its `NSTextStorage` instance. `NSTextStorage` is a subclass of `NSMutableAttributedString`, which implements the text styling features: it allows *attributes* (such as foreground color or font) to be applied to (or removed from) arbitrary sections of the string.

In addition to setting the target text view widget, our highlighter class exposes methods in its public interface for activating and deactivating it: an "active" highlighter listens to events in the text view widget and applies highlighting automatically, while inactive highlighters have to be manually told to parse and highlight. In accordance with one of our functional requirements, the highlighter also exposes a way to set the order in which different language elements are highlighted, as well as the

[9] A metacharacter is not considered part of the document's content.

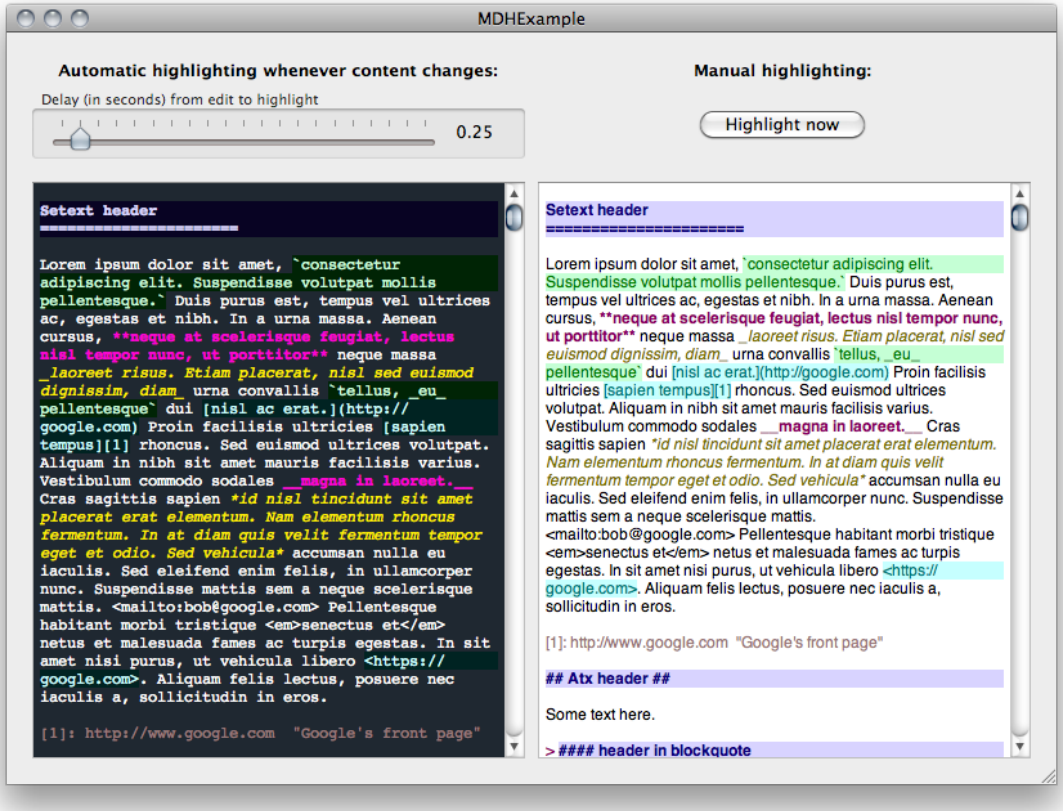

Figure 5: Screenshot of a Cocoa prototype using the highlighter for two separate NSTextViews, with different visual styles

visual styles that are used to highlight them.

Figure 5 shows a screenshot of a proof-of-concept prototype that uses the Cocoa highlighter classes to highlight Markdown content in two separate `NSTextView` instances, with different styles.

5.2.1 Techniques for Optimizing User Interface Response

Since the actual highlighting of the GUI text widget must be done in the user interface thread, we want it to be as fast as possible so that the user experience does not suffer due to slow interface response. In this section we go through a few different ways we try to keep this from happening.

When the highlighter receives a notification of the text in the editor widget changing, it does not begin parsing immediately. Instead it starts a timer (or resets it, if it was already running) and begins parsing only when the timer finishes. We do this so that multiple quick sequential changes (for example due to the user typing text into the control) would not each trigger parsing and highlighting and thus slow down the user interface. The highlighter class implements a method to change the length of this time interval.

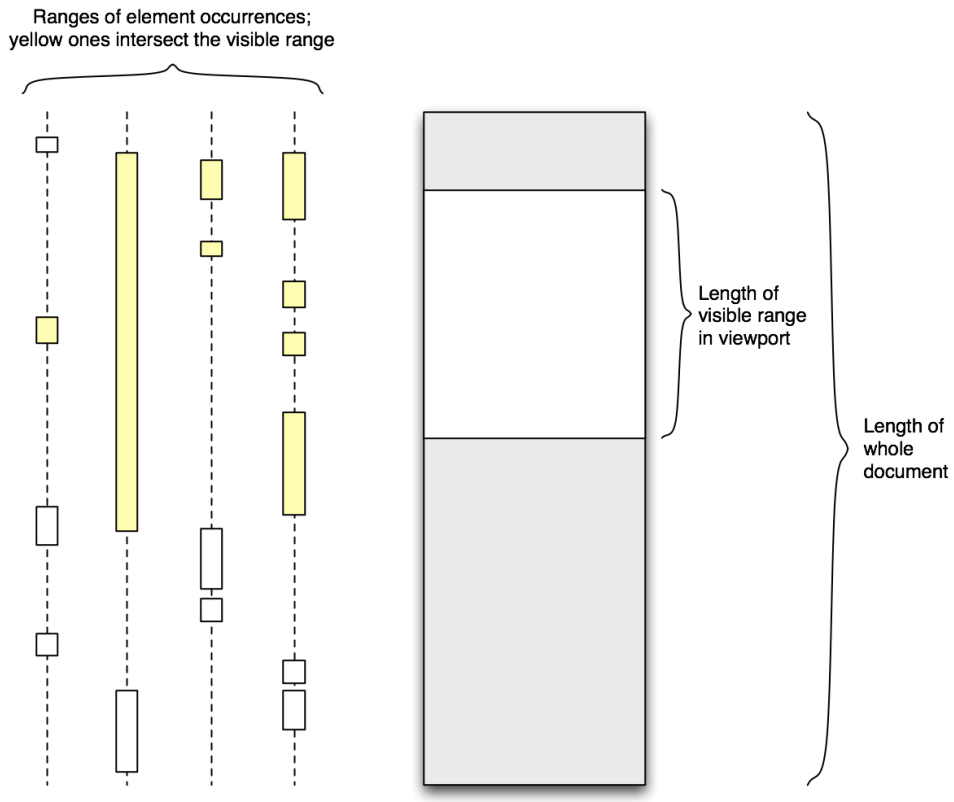

Figure 6: We want to find element occurrences whose ranges intersect with the visible range of the input text

Since the process of parsing the text does not require access to anything in the GUI, we can do it in a background thread. Each highlighter instance has one thread which it uses to run the parser, and in case of new changes in the text happening while the parser is still processing the previous version, the highlighter simply discards the output of the parser when it finishes, and runs it again on the changed text. Since the parser generated by greg is re-entrant, multiple parser threads (each owned by a different highlighter class instance) can run at the same time within the same process.

When the parser has completed and we want to highlight the text editor widget contents, instead of processing the complete text we process only the part that is currently visible to the user (see figure 6). This is a good optimization because it significantly reduces the amount of work needed — the longer the text is the more useful this optimization becomes — but it also requires the highlighter to know when the range of visible text changes in the widget so that it can highlight this new range. This is possible in Cocoa: the class `NSScrollView` (instances of which enclose `NSTextViews`) sends notifications when its bounds change, and the highlighter can subscribe to receive them. Since we don't want to run the parser needlessly whenever the text editor widget is scrolled or resized we make the highlighter class cache the latest data structure produced by the parser.

6 Evaluation of Implementation

In this section we evaluate the highlighter against the requirements we outlined in section 3. All of the functional requirements have been fulfilled by the implementation so this section focuses on evaluating the non-functional requirements.

6.1 Evaluation of Correctness

The correctness property of the highlighter should correspond almost fully to that of peg-markdown's; the highlighter is designed to behave exactly like the compiler it is based on and the only known regression is that the title in a reference definition is not highlighted if it is on the last line of the input (that is, if the line does not end in a "newline" character). In other words there is one known minor defect that is exhibited in rare circumstances. This is a violation of our correctness requirement but since the magnitude is so low (both in terms of the problem it poses to users and its architectural significance) we consider the correctness of the current implementation acceptable.

6.2 Evaluation of Efficiency

We use the same Markdown documents and measurement methods as in section 4.4 to compare the execution time and peak heap memory usage between the peg-markdown compiler and the highlighter we created using its parser. Table 5 shows the execution times (averaged over 100 iterations) and table 6 the memory usage peaks.

Table 5: Comparison of compiler and highlighter: average compilation/highlighting times (in seconds)

Program	normal.md (x 20)	eccentric.md (x 130)
peg-markdown: compiling	0.079	0.126
highlighter: parsing and highlighting	0.064	0.125
highlighter: parsing only	0.052	0.089

As noted in section 4.4, peg-markdown is already much faster than most other Markdown compilers, and as seen in table 5 and figure 7, for the tested two inputs our highlighter is at least as fast.[10] Note that in this evaluation we measure the time it takes to parse and highlight the complete input text, even though in normal usage we would only highlight the range of text visible to the user — we do this in order to make the comparison to compiling the whole document fair. The table also shows measurements for parsing without highlighting, which tells us that the lion's share of

[10]The highlighter's execution time is measured using a small command-line program that reads a file from disk, feeds its contents to the parser and highlights an *NSMutableAttributedString* based on the output. It measures and finally outputs the time interval between start of parsing and end of highlighting.

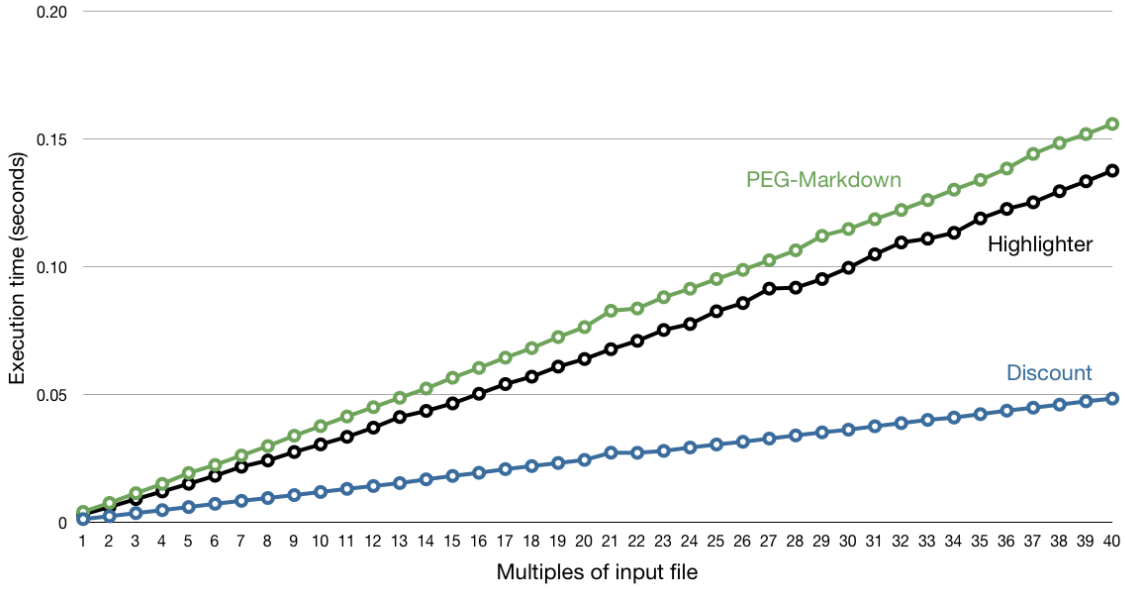

Figure 7: Growth of highlighter execution time for longer inputs, in comparison to the peg-markdown and Discount compilers

execution time is spent in parsing. The figure shows how the highlighter's execution time grows quite linearly as the size of the input increases.[11]

This result is not enough to make the highlighter as fast as the Discount compiler but certainly enough to make it useful and fulfill our speed requirement.

Table 6: Comparison of compiler and highlighter: peak heap memory usage (in MB)

Program	normal.md (x 20)	eccentric.md (x 130)
peg-markdown: compiling	4.38	6.70
highlighter: parsing only	1.53	5.80

In 4.6 we posited that peg-markdown's code generation might be responsible for a large part of its memory usage, and in table 6 we see how the highlighting parser indeed uses noticeably less memory than the full compiler.[12] The decrease seems to vary a lot depending on the type of input: for normal.md it is 65% but for eccentric.md only 13%. Overall the memory usage is acceptable and easily fulfills our requirement.

[11]The inputs are multiples of "normal.md" just like in 4.4.1.

[12]The highlighter parser's memory usage is measured from a small command-line program that reads a file from disk, feeds its contents to the parser and prints out the indexes of element occurrences.

6.3 Evaluation of Portability

In order to evaluate the portability of the parser, we need to determine whether it compiles and runs (along with its dependencies) on the Mac OS X, Linux and Windows operating systems.

One problem for portability is peg-markdown's dependency on the glib2 library. For our highlighting parser, this dependency is easily removed and replaced with custom C code. The parser thus has no third-party dependencies, which makes porting to other systems much easier.

We have originally written the parser on a Mac OS X system. Due to the common Unix heritage of OS X and Linux, the use of the GNU C compiler (GCC), and the fact that the parser was written with ANSI/ISO C89 with GNU extensions, no porting effort is necessary to get it to compile and run on Linux. Since the GNU extensions used in the code are also part of the C99 standard, any C compiler that supports C99 should work.

Since the parser has no third-party dependencies and can be compiled with GCC, the *MinGW*[13] environment can be used to compile it for Windows systems. More important than compatibility with MinGW, though, is the ability to compile the parser with the *Microsoft Visual C++ Compiler* (MSVC), since that is what developers of Windows programs are more likely to actually use.

The problem with MSVC is that it has no support for the C99 standard, nor for any of the GNU extensions to C89. The obvious route to compatibility, then, is to remove from the parser all use of features that are not specified in C89, but this would deprive us of all the nice things that C89 does not support (such as declaring variables just-in-time or preprocessor macros with variadic arguments). Instead of "moving down" to the level of C89 we can "move up" to the level of C++ with a few simple changes to our code, and have it compile as C++ in MSVC. In the particular case of our parser, these changes include renaming variables so that C++ reserved names (such as "new") are not used, explicitly casting values in many cases where MSVC expects it, and not including `stdbool.h` if compiled as C++ (we can determine this with `#ifdef __cplusplus`).

Since the parser has no third-party dependencies and can be compiled and executed on OS X, Linux and Windows systems, it fulfills our portability requirement.

6.4 Evaluation of Embeddability

We want the parser to be easily embeddable into applications written with different GUI application frameworks, and in our requirements we specified that demonstrating this quality using two different frameworks is sufficient. We also want the Cocoa highlighter implementation to be embeddable into arbitrary third-party applications (that are written with Cocoa, of course), and this quality can likewise be demonstrated by embedding the highlighter into two different applications. We discuss these two aspects separately in the following subsections, and focus in both cases on how easy the integration is to perform.

[13] MinGW is a development environment that contains a port of the GNU C compiler that can be used to compile native Windows programs. See `http://www.mingw.org` for more information.

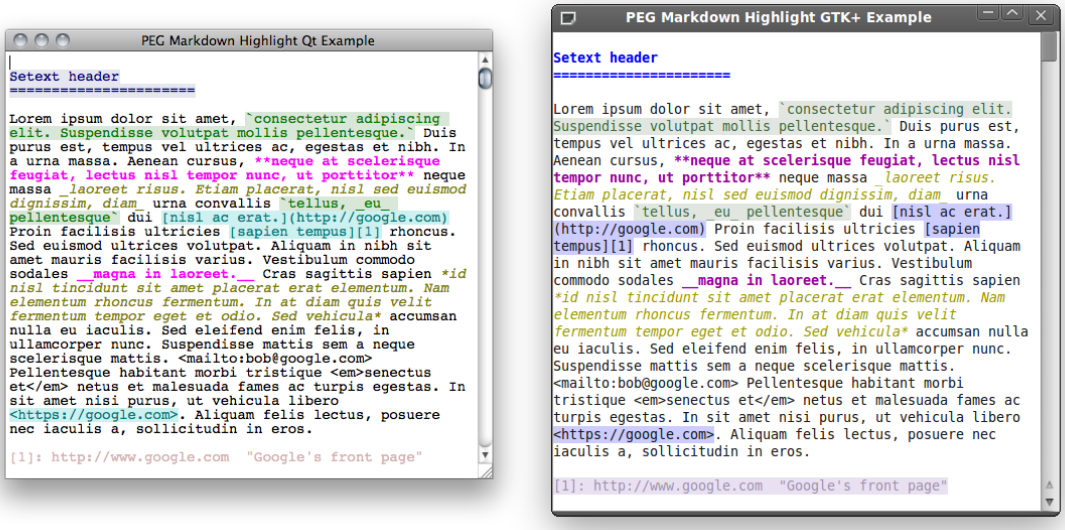

Figure 8: Screenshots of the Qt (on the left, running on Mac OS 10.6) and GTK+ (on the right, running on Ubuntu Linux 9.04) prototypes

6.4.1 Using the Parser with Different GUI Application Frameworks

The highlighter classes are implemented with the Cocoa framework, and we have also integrated the parser into proof-of-concept programs using the GTK+ 2 and Qt 4.7 frameworks (figure 8 shows screenshots of the latter two). In section 5.2 we discussed the implementation of the highlighter in Cocoa, and here we look at some issues regarding parser embeddability in the other two proof-of-concept prototypes.

GTK+ is an open-source GUI application framework written in the C language. Qt is likewise open-source but written in C++. Since our parser is written in C, it can be statically compiled into and invoked from both GTK+ and Qt programs. The critical parts in using the parser to highlight text editor GUI widgets are the facilities these widgets expose for controlling the visual styles of their textual contents.

The GTK+ text editor widget class, `GtkTextView`, uses a `GtkTextBuffer` instance to store and manage access to its contents. This buffer class provides easy access to the contained text and makes it possible to apply *tags* that describe formatting decisions (such as foreground color, or bold font state) to arbitrary positions in the text. [2] These facilities are easy to use and work well for our purposes.

Qt includes the `QSyntaxHighlighter` class, which is designed as a simple way to add syntax highlighting of custom grammars into text editor widgets (instances of `QTextEdit`). Unfortunately the fundamental design of this built-in syntax highlighting mechanism dictates that highlighting be performed separately for each individual "block" of text (that is, paragraphs separated by blank lines). The class makes it possible to inspect the state of the previous block as a way of handling highlighting rules that span multiple blocks, but exposes no way to consider the whole document at once (which is what our solution requires). [5] Following the same design, `QTextEdit` uses a `QTextDocument` instance to store and manage access to its contents, and the

interface this document class provides is also fundamentally based on "blocks". There is thus a mismatch between the requirements of our parser and Qt's text editor implementation.

Each block (represented by a `QTextBlock` instance) in a `QTextDocument` has its own `QTextLayout` instance that handles the visual display properties of the block's contents. In order to set a specific visual style to a given range of text we need to ask the document object for all of the blocks that our range spans through, and for each of these blocks apply formatting rules via their text layout objects while translating the start and end indexes of our original span to the internal indexes of each block object.

Using our parser to highlight text in GUI text editor widgets is quite straightforward if you are using Cocoa or GTK+, and possible — albeit complex — if you are using Qt. The complexities in Qt integration are due to an unfortunate mismatch between fundamental design decisions made in both the Qt text editing subsystem and our parser, but neither can be particularly blamed for this (we are confident that our parser's design is justified, given the use case, and we trust that the same applies for Qt). Since we have successfully used the parser with three different GUI application frameworks, this part of the embeddability requirement is fulfilled.

6.4.2 Integrating the Cocoa Highlighter into Different Applications

We embed our highlighter into two different Cocoa applications: *Notational Velocity* and *MarkdownLive*. The versions of these applications are listed in appendix 9.1.

Notational Velocity is an open-source GUI application for Mac OS X for writing clear text documents, or notes, and quickly searching through them. Figure 9 shows our highlighter integrated into it.

Unlike many other OS X applications that deal with documents of some kind, the main user interface of Notational Velocity is contained in a single window. An instance of the "application controller" class (`AppController`) controls the main functionality of the program, and most notably the single `NSTextView` instance that displays the contents of the selected note. We can use our highlighter in this application by instantiating it in the application controller class and using the interface it provides for connecting it to the text view widget. A method in `AppController` is responsible for switching between displayed notes, so there we also need to determine whether the newly selected note is Markdown-formatted (by inferring it from the file extension) and activate or deactivate the highlighter accordingly.

The Notational Velocity project uses quite strict compiler settings (such as enabling many optional warnings and treating all warnings as errors), which means that our parser and highlighter need to abide by the same rules.

MarkdownLive is an open-source GUI application for Mac OS X for editing Markdown documents with an integrated web browser preview of the translated HTML version. Figure 10 shows our highlighter integrated into it.

As opposed to Notational Velocity, each open document in MarkdownLive has a window (and an `NSTextView` instance) dedicated to it. The application is based on the Cocoa document framework and almost all of its code is in a single subclass of `NSDocument`. [11] Following the existing design of the application (where the HTML

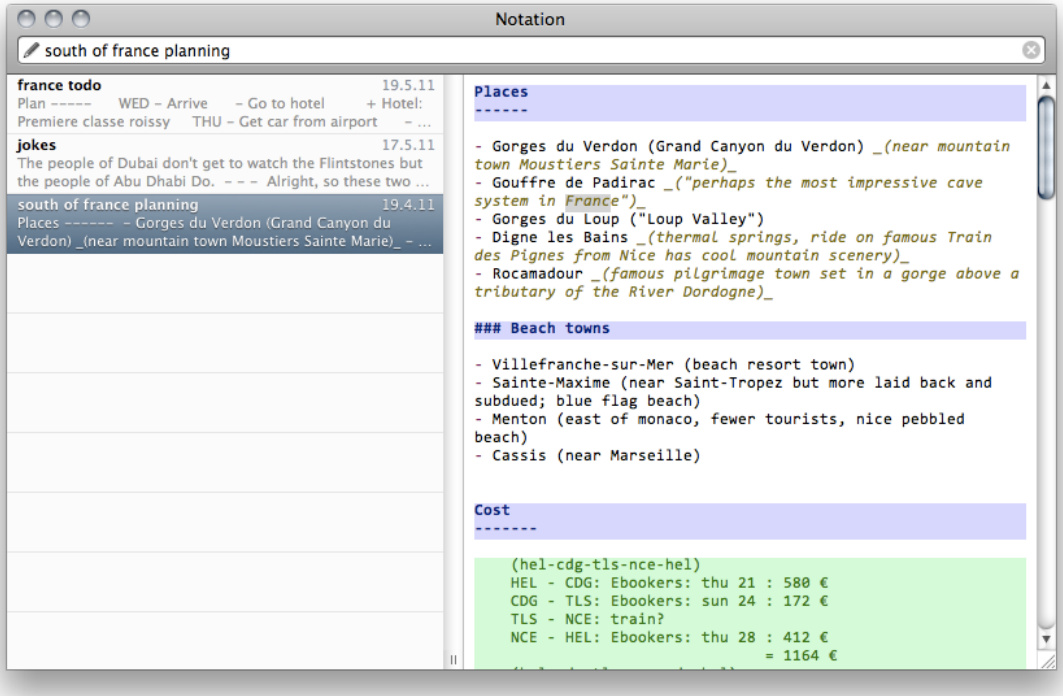

Figure 9: Screenshot of our highlighter integrated into the Notational Velocity application

preview functionality is in the `NSDocument` subclass) we can integrate our highlighter there simply by instantiating it and using the interface it provides for connecting it to the text view widget in the document window. The re-entrant nature of the highlighter's parser is necessary in cases like this where multiple parser threads need to be able to run at the same time.

Based on these two cases it seems that our Cocoa highlighter classes have the necessary functionality and an appropriate design for integrating into third-party applications, "document-based" or not.

7 Directions for Future Work

As is with Markdown compilers, the main focus of future work for our highlighter will be finding and fixing incompatibilities with the language syntax description and the Markdown.pl compiler. One way to do this in a systematic manner would be to gather a large corpus of Markdown documents from the web and manually sift through the highlighter's results for them in order to find problems. For every issue that is found, we would add a corresponding test case to the MDTest suite and only then fix the bug. When the test cases are in place, we will have a system for monitoring possible future regressions.

Another avenue of improvements is in optimizing the highlighter to perform its

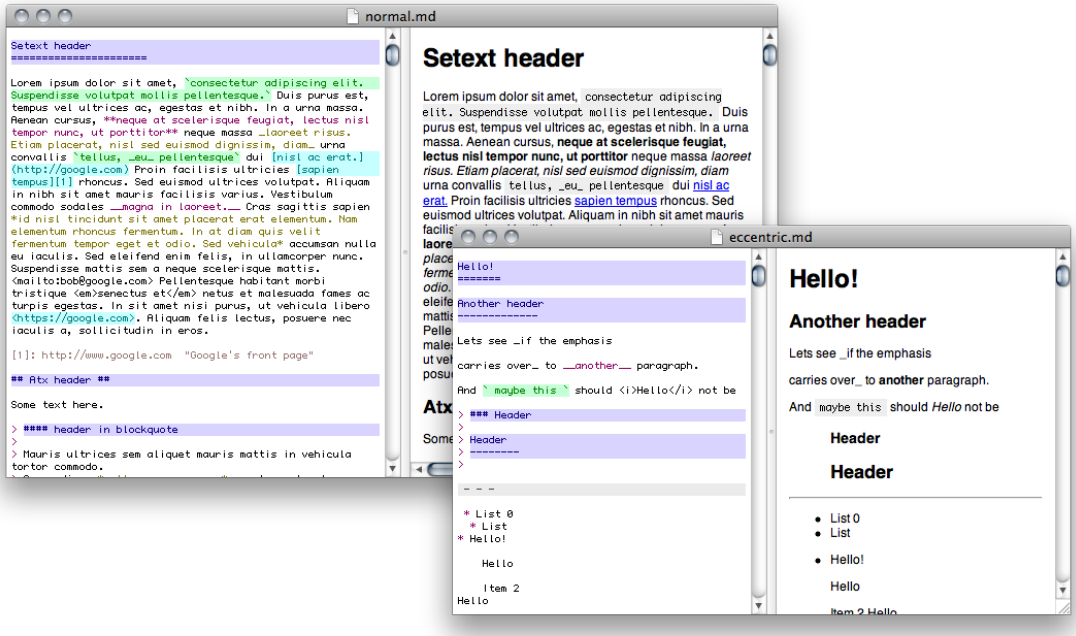

Figure 10: Screenshot of our highlighter integrated into the MarkdownLive application, displaying two separate document windows

job even faster, with a particular focus on the work that must be done in the user interface thread. The following subsections discuss possible optimizations to parsing and highlighting.

7.1 Possible Parsing Optimizations

The output generated by greg from our PEG grammar is a recursive descent parser. For grammars that are not LL(1) recursive descent parsing requires backtracking, which lowers performance, and in the worst case can lead to exponential complexity. *Packrat* parsing is a method that ensures linear complexity for these kinds of grammars, at the expense of much higher memory requirements. [3, pp. 1–2] [6, pp. 14–15]

Although superlinear complexity might be a concern for recursive descent parsers in general, we have shown that our parser exhibits close to linear performance for the tested Markdown inputs. This, along with the increased memory consumption, clearly diminish the usefulness of the packrat approach in our case. It is nevertheless a technique that is good to keep in mind, especially in case a third party adds such capability to peg/leg or greg in the future.

7.2 Possible Highlighting Optimizations

Our implementation finds the ranges of element occurrences that intersect the visible range of text (see figure 6) through a linear search, so the time complexity is *O(n)*,

where n is the length of the element list. We order the occurrences by their starting position, so if we denote the length of the visible range of text with v and the length of the whole text with t, the best case is when the viewport is at the very top (we only need to search the first elements until we find one that starts after v) and the worst when the viewport is at the very bottom (we need to search the whole list of elements). Note that if we order the occurrences by their ending position, the situation will be symmetric and inverted. There are a few possibilities for optimizing this that we may investigate in the future.

One option is to order the element occurrences in the lists by their *midpoint* instead of starting point. This would not improve the asymptotic complexity of the algorithm — it would still be linear — but it would even out the difference between the best and worst cases: the search range would always be $\frac{t+v}{2}$, regardless of viewport position (we would always search the range between the midpoints of the sections in the text before the viewport start and after its end). Using linked lists to store the element occurrences diminishes the usefulness of this approach, though — being able to calculate the search range beforehand is fully useful only if you have random access into the searched list.

Another option is to use an *interval tree* data structure to store the element occurrences; finding ones that intersect the visible range could then be done in *O(lg n)* time. Even though this approach would improve the asymptotic complexity of the search algorithm we must note that building this data structure — which can be done at the time of parsing and thus in a background thread — takes *O(n lg n)* time. [4, pp. 348–351] One other practical consideration is that Markdown documents tend to be quite short, which means that for common use cases this approach might actually be slower than the naïve linear one.

8 Conclusion

We have evaluated five candidate parsers in existing Markdown compilers in order to determine which one of them would provide the best starting point for our syntax highlighter implementation, and chosen peg-markdown due to its high execution speed, adequate correctness and memory usage, and particularly due to its use of a declarative PEG grammar, which makes it very easy to modify. We have then discussed the details of adapting peg-markdown's parser for syntax highlighting as well as the design of Objective-C classes for highlighting text editor widgets in Cocoa applications. Finally, we have evaluated the syntax highlighter implementation and determined that it performs the functions we want and fulfills all of our desired quality attributes.

Applications that allow users to edit Markdown documents will benefit from using our solution if accurate syntax highlighting is desired: it is efficient enough for this purpose and easy to integrate — and most important, it handles Markdown's context sensitivity as well as existing compilers do. Our highlighter is licensed under both the MIT License and the GNU GPL version 2 (or any later version), which means it can be used in both open-source and proprietary programs.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, second edition, 2007.

[2] Vijay Kumar B. Formatted text in GtkTextView. `http://www.bravegnu.org/gtktext/x113.html`. Accessed May 23, 2011.

[3] Ralph Becket and Zoltan Somogyi. DCGs + memoing = packrat parsing; but is it worth it? In *Proceedings of the Tenth International Symposium on Practical Aspects of Declarative languages*, 2008.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[5] Nokia Corporation. QSyntaxHighlighter class reference. `http://doc.trolltech.com/4.7/qsyntaxhighlighter.html`. Accessed May 23, 2011.

[6] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[7] Michel Fortin. Mdtest's git repository. `http://git.michelf.com/mdtest/`. Accessed March 8, 2011.

[8] John Gruber. Daring fireball: Markdown syntax documentation. `http://daringfireball.net/projects/markdown/syntax`. Accessed March 12, 2011.

[9] John Gruber. Trailing spaces in code block. `http://six.pairlist.net/pipermail/markdown-discuss/2006-June/000079.html`. Accessed March 8, 2011.

[10] Apple Inc. Categories and extensions. `http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocCategories.html%23//apple_ref/doc/uid/TP30001163-CH20`. Accessed May 18, 2011.

[11] Apple Inc. Document-based application architecture. `http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/Documents/Concepts/OverviewDocArchitecture.html%23//apple_ref/doc/uid/20000023-BAJBBBAH`. Accessed May 29, 2011.

[12] Lablua, Departamento de Informática, PUC-Rio. Lua: About: Why choose lua? `http://www.lua.org/about.html#why`. Accessed March 6, 2011.

[13] John McFarlane. Peg-markdown readme. `https://github.com/jgm/peg-markdown#readme`. Accessed March 6, 2011.

[14] John McFarlane. Re: lunamark - markdown in lua using a peg grammar. `http://www.mail-archive.com/markdown-discuss@six.pairlist.net/msg01661.html`. Accessed March 8, 2011.

[15] John McFarlane <jgm#at#berkeley.edu>. Peg-markdown: Logic behind alphanumeric & nonalphanumeric definitions. E-mail.

[16] Chad Nelson. Cpp-markdown readme.

[17] David Parsons. Discount. http://www.pell.portland.or.us/~orc/Code/discount/#Source.Code. Accessed March 6, 2011.

[18] David Parsons. Parser question + markdown "wysiwyg". http://six.pairlist.net/pipermail/markdown-discuss/2010-April/001767.html. Accessed March 8, 2011.

[19] Unicode Consortium. The unicode standard version 6.0 – core specification. http://www.unicode.org/versions/Unicode6.0.0/ch03.pdf#G7404, 2011.

[20] Unicode, Inc. The unicode® standard: A technical introduction. http://unicode.org/standard/principles.html, 2010. Accessed March 12, 2011.

[21] Unicode, Inc. FAQ – UTF-8, UTF-16, UTF-32 & BOM. http://unicode.org/faq/utf_bom.html#bom5, 2011. Accessed June 2, 2011.

[22] Amos Wenger. greg. https://github.com/nddrylliog/greg#readme. Accessed March 16, 2011.

[23] Wikipedia: Markdown users. http://en.wikipedia.org/wiki/Markdown#Markdown_users. Accessed March 12, 2011.

9 Appendix

9.1 Version Information for Software and Tools

Table 7 details versions of infrastructure and tools used for the evaluation of existing compilers and table 8 the versions of the compilers themselves. Versions of frameworks and applications used in the evaluation of the implementation are shown in table 9.

9.2 Test File “normal.md”

The following are the contents of the test file “normal.md”:

```
1
2 Setext header
3 ======================
4
5 Lorem ipsum dolor sit amet, ‘consectetur adipiscing elit.
  Suspendisse volutpat mollis pellentesque.‘ Duis purus est,
  tempus vel ultrices ac, egestas et nibh. In a urna massa. Aenean
   cursus, **neque at scelerisque feugiat, lectus nisl tempor nunc
  , ut porttitor** neque massa _laoreet risus. Etiam placerat,
  nisl sed euismod dignissim, diam_ urna convallis ‘tellus, _eu_
```

Table 7: Versions of infrastructure and tools used in the evaluation of existing parsers

Tool	Version	Other Information
Apple Macbook	Macbook2,1	2 GHz Core 2 Duo, 2 GB RAM (white, '06)
Mac OS X	10.6.6	
GCC	4.2.1	Apple Inc. build 5666 (System standard)
Perl	5.10.0	System standard
Lua	5.1.4	From MacPorts
LuaJIT	2.0.0-beta6	
LuaSocket	2.0.2	From MacPorts
Valgrind	3.6.1	From MacPorts
Ohcount	3.0.0	From MacPorts
MDTest	1.1	Git: `6321a5efaf4490cfde697c6e12bf33d62037e29e`
MarkdownTest	1.0.3	

Table 8: Versions of evaluated Markdown compilers

Compiler	Version	Other Information
Discount	2.0.8	
peg-markdown	0.4.12	Git: `d57e706b7082d2c693314235ea90542da04d39c8`
Cpp-Markdown	1.0	
Lunamark	?	Git: `6877d4bb4a98ddbf9eabc25d3d7cdfb964ccb961`
Markdown.lua	0.32	
Markdown.pl	1.0.2b8	

Table 9: Versions of frameworks and applications used in the evaluation of our highlighter

Software	Version	Other Information
Qt	4.7.1	Used on Mac OS X version 10.6.6
GTK+	2.16.1	Used on Ubuntu version 9.04
Notational Velocity	2.0 β5	Git: `4e544ea41f46aa58ea9d55a7a2c7c2351a053f50`
MarkdownLive	1.5.1	Git: `ae03663ae9c4bf9ca1af001cc9394534818f8b0c`

```
pellentesque‘ dui [nisl ac erat.](http://google.com) Proin
facilisis ultricies [sapien tempus][1] rhoncus. Sed euismod
ultrices volutpat. Aliquam in nibh sit amet mauris facilisis
varius. Vestibulum commodo sodales __magna in laoreet.__ Cras
sagittis sapien *id nisl tincidunt sit amet placerat erat
elementum. Nam elementum rhoncus fermentum. In at diam quis
velit fermentum tempor eget et odio. Sed vehicula* accumsan
nulla eu iaculis. Sed eleifend enim felis, in ullamcorper nunc.
Suspendisse mattis sem a neque scelerisque mattis. <mailto:
bob@google.com> Pellentesque habitant morbi tristique <em>
senectus et</em> netus et malesuada fames ac turpis egestas. In
sit amet nisi purus, ut vehicula libero <https://google.com>.
Aliquam felis lectus, posuere nec iaculis a, sollicitudin in
eros.

[1]: http://www.google.com  "Google's front page"

## Atx header ##

Some text here.

> #### header in blockquote
>
> Mauris ultrices sem aliquet mauris mattis in vehicula tortor
commodo.
> Suspendisse *nulla neque, ornare* non laoreet vel, **gravida
vitae nisi.
> Etiam odio lacus, ‘pellentesque‘ imperdiet vehicula ut,**
consectetur nec
> sapien. Ut varius ante vitae nisl commodo ultricies. Cras
sapien dolor,
> facilisis in tincidunt vel, congue eu risus. Aliquam erat
volutpat.
>
>     Code in blockquote
>     Code in blockquote
>
> Pellentesque ut tincidunt metus. Praesent egestas
__pellentesque blandit.__
> Pellentesque adipiscing _mollis eros._ Phasellus cursus metus
sed magna
> lobortis porttitor.
>
> - Lorem ipsum dolor sit amet.
>     - ‘consectetur adipiscing elit‘.
>     - Cras quam [lectus][1], blandit et mattis
>       a, volutpat quis tortor.
> - Fusce vel ante dui. Ut eget orci eget lacus
>   ultricies [faucibus non a
```

```
>   dolor](http://google.com).
>     - Mauris vestibulum lacus quis quam tempor ut
>       rhoncus odio vulputate.

![image alt](http://google.com/ "Image title")

    Nulla euismod, nibh non dignissim dapibus, dui augue
    varius elit arcu vitae turpis. Morbi tempus, sem ac

* blandit ante, **et** consectetur
  ante felis eleifend eros.

  asdj lsadj `laskjd lkasdj dd
  sadj sladj` lasdj *lsdj lsdkj*

* Mauris _lacinia_ dolor nec
  eros hendrerit at ultricies
  odio egestas.
    * Vestibulum **quis sodales dolor**.
    * Duis *adipiscing sodales* nibh sed venenatis.
        1. Proin condimentum `tincidunt leo vel`!
        1. aliquam. __Duis tincidunt [nunc](http://google.com)
        risus,
           sed dignissim__ quam. *Morbi* dapibus dictum
* elit, vel pulvinar dui porttitor varius. Curabitur eu tortor
magna.
* Aliquam *risus <http://google.com>
  risus* purus, blandit <http://bing.com>
  quis dictum id, eleifend in dolor.

### Atx header

<div id="asdsad">
Nunc hendrerit sagittis ornare. Mauris dui ipsum, imperdiet in
accumsan ac, euismod ac sem. Morbi eu turpis vitae purus euismod
 pharetra. Duis dui lorem, rutrum vitae mollis a, malesuada sit
amet &rarr; massa. &copy; Donec sem nibh, interdum nec aliquam <
em>a</em>, condimentum sed massa. Fusce sed nunc dui. Nullam
venenatis rutrum commodo. Maecenas dolor nisi, scelerisque sed
luctus vel, malesuada at tortor. <a href="google.com">
Pellentesque sagittis ultrices</a> tincidunt. Ut placerat dolor
id urna aliquet fermentum. Nullam adipiscing neque interdum &
trade; dui imperdiet id viverra risus porta. Morbi quis felis
libero. Vestibulum erat dolor, aliquam pharetra lobortis quis,
tristique nec urna.
</div>

1. blandit ante, **et** consectetur
   ante felis eleifend eros.
```

```
    asdj lsadj `laskjd lkasdj dd
    sadj sladj` lasdj *lsdj lsdkj*

1. Mauris _lacinia_ dolor nec
   eros hendrerit at ultricies
   odio egestas.
    1. Vestibulum **quis sodales dolor**.
    1. Duis *adipiscing sodales* nibh sed venenatis.
        * Proin condimentum `tincidunt leo vel`!
        * aliquam. __Duis tincidunt [nunc](http://google.com)
        risus,
          sed dignissim__ quam. *Morbi* dapibus dictum
1. elit, vel pulvinar dui porttitor varius. Curabitur eu tortor
magna.
1. Aliquam *risus <http://google.com>
   risus* purus, blandit <http://bing.com>
   quis dictum id, eleifend in dolor.

Nunc erat mauris, fringilla quis scelerisque eget, pharetra a
odio. Sed vulputate turpis et sem molestie vehicula et non neque
. Aliquam nulla urna, iaculis sed pharetra in, lacinia vitae
tortor.

- Lorem ipsum dolor sit amet, `consectetur adipiscing
  elit. Suspendisse volutpat mollis pellentesque.`
- Duis purus est, tempus vel ultrices ac, egestas et nibh.
    - In a urna massa.
    - Aenean cursus, **neque at scelerisque feugiat,
      lectus nisl tempor nunc, ut porttitor** neque massa
      _laoreet risus.
      Etiam placerat, nisl sed euismod dignissim, diam_ urna
      convallis `tellus, _eu_ pellentesque` dui [nisl ac
      erat.](http://google.com)

      Proin facilisis ultricies
      [sapien tempus][1] rhoncus. Sed euismod ultrices volutpat.
      Aliquam in nibh sit amet mauris facilisis varius.

        - Vestibulum commodo sodales __magna in laoreet.__ Cras
          sagittis sapien *id nisl tincidunt sit amet placerat
          erat elementum.
            - Nam elementum rhoncus fermentum. In at diam quis
              velit fermentum tempor eget et odio. Sed vehicula*
              accumsan nulla eu iaculis.
                - Sed eleifend enim felis, in ullamcorper nunc.
                    - Suspendisse mattis sem a neque scelerisque
                      mattis.
```

```
111                             - <mailto:bob@google.com> Pellentesque
                                habitant
112                               morbi tristique <em>senectus et</em>
                                  netus et
113                               malesuada fames ac turpis egestas.
114                                 - In sit amet nisi purus, ut
                                    vehicula libero
115                                   <https://google.com>. Aliquam
                                      felis lectus,
116                                   posuere nec iaculis a,
                                      sollicitudin in eros.
117
118
119 Praesent ultricies molestie semper. Nam malesuada mi quis nisi
    molestie dignissim. Integer lectus nibh, tempor non ornare ut,
    pretium non felis. Praesent congue euismod aliquet. Aenean erat
    felis, lobortis sed dignissim et, lobortis ac arcu. Donec sit
    amet erat eget nunc consequat gravida. Aenean enim purus, congue
     id venenatis in, posuere et risus. Nulla nunc nibh, luctus quis
     lacinia vitae, congue convallis eros. Nunc feugiat luctus
    bibendum. Vivamus sit amet ipsum nunc, at vulputate purus.
    Mauris elementum nibh commodo nisi rhoncus quis pharetra est
    pellentesque. Maecenas est mi, pretium eu euismod quis,
    tristique id turpis.
120
121 Setext h2
122 ---------------------
123
124 Maecenas accumsan rhoncus erat, ac commodo urna viverra in.
    Aliquam nisi arcu, semper vitae porta quis, sollicitudin vel
    metus. Duis sit amet dui nisi. Phasellus posuere adipiscing nunc
     nec dignissim. Ut id interdum diam.
```

9.3 Test File “eccentric.md”

The following are the contents of the test file “eccentric.md”:

```
 1
 2 Hello!
 3 =======
 4
 5 Another header
 6 --------------
 7
 8 Lets see _if the emphasis
 9
10 carries over_ to __another__ paragraph.
11
12 And ‘ maybe this ‘ should <i>Hello</i> not be
```

```
13
14 > ### Header
15 >
16 > Header
17 > --------
18 >
19
20  - - -
21
22  * List 0
23   * List
24 * Hello!
25
26     Hello
27
28     Item 2
29 Hello
30
31     Hibuli habuli!*Code here*
32
33   * Item 3
34
35  * Hello again
36
37             Hassd *code*
38
39         Code here
40
41 <img src="nav_logo6.png" />
42
43  1. Stuff
44
45 Hello
46
47         Markdown FTW!
48
49
50 1986\. What a great season. A link here: <http://www.example.com
   /> that should be automatically detected.
51
52 Stuff *here!*. This is pretty cool.
53
54 * Hello
55     * Booboo
56
57 What [up to the ][name] cats n dogs :) He*ll*o!! *Some
   emphasized* stuff here. A * star * has [born]. This * i dont
   know what*will happen.
58
```

```
1. Stuff
1. Hello
9. Again
3211233. Stuff

This is [an example](http://example.com/ "Title") inline link.

![][googleimg]

[googleimg]: http://www.google.com/images/nav_logo6.png
"Google!"

‘‘Hello **‘strong text** here‘‘

> These * should * not \*be\* selected. This* neither! *should be. This *neither should\* be*

# 2nd Headline

Yeah.

[name]: http://google.com
[foo1]: http://example.com/  "Optional Title Here"
[foo3]: http://example.com/
  "’Optional Title Here"


    Some code


Third headline
--------------------
Some stuff here...
```