Course Project, Part 1

Due Sept. 21

September 10, 2018

1 Description

Over the course of the semester, you (along with a partner) will develop an interpreter for interactive fiction. Interactive fiction (IF) is a text-based form of entertainment in which a reader is given passages of text interspersed with choices. In a sense, these resemble the "Choose Your Own Adventures" that were once popular; however, IF may be more sophisticated in that early choices in the story may influence later passages or choices.

In this assignment, you will write code to parse out the different parts of a work of interactive fiction. Note that while this assignment is *individual*, much of the rest of the project will be completed with a partner, so getting too far ahead in implementing the project at this point may result in wasted effort.

2 Specifications

Your IF interpreter will be based on a limited subset of Harlowe (`https://twinery.org/wiki/harlowe:reference`), a common format for IF.

3 Tokenization

When interpreting complex text input, it is common to first put that input through a process called *tokenization*, which splits up the input into meaningful chunks (such as words or lines) that can then be interpreted.

Your goal for this assignment is to write two classes that can tokenize interactive fiction stories, a StoryTokenizer and a PassageTokenizer. Your StoryTokenizer should be able to extract *passage tokens* from its input, where each passage token represents one passage in the story. The PassageTokenizer should be able to extract section tokens from a passage, representing the important parts that comprise that passage. Your code should be able to recognize a total of 8 different kinds of section tokens: *links*, *commands*, *blocks*, and *text*, where there are 5 different varieties of command tokens: *go-to*, *set*, *if*, *else-if*, and *else*. These tokens will be described in more detail in the following sections.

3.1 Passage tokens

Interactive fiction works are divided into passages, which appear in the HTML tags `<tw-passagedata>`. Each passage will start with `<tw-passagedata ...>` and will end with `</tw-passagedata>`. In addition to starting with `<tw-passagedata`, the opening tag will specify some attributes, and the body of the passage will be between the opening and closing tags.
Example passage:

```
<tw-passagedata pid="1" name="start" tags="" location="100,100">
The body of the passage will be here.
</tw-passagedata>
```

Your StoryTokenizer should have two member functions: `hasNextPassage` and `nextPassage`. As can be inferred from the name, `hasNextPassage` returns whether the story contains another passage (i.e., one that has not been read in yet), while `nextPassage` returns a PassageToken object describing the passage. It should also have a constructor that accepts a string.

PassageTokens should have one member function, `getText`, as well as an appropriate constructor. The `getText` member function should return the text of the passage (between the starting tag `<tw-passagedata ...>` and the ending tag `</tw-passagedata>`). In the example above, this text would contain "The body of the passage will be here.", with newlines before and after. An invalid PassageToken (e.g., the return result of `nextPassage` when there are no more passages) should return an empty string as its text.

3.2 Section tokens

While stories are divided into passages, it's important to be able to separate out the different parts of the passage. Your PassageTokenizer should implement `nextSection` and `hasNextSection`, which return the next SectionToken and whether the passage contains another section, respectively. You should have a PassageTokenizer constructor that accepts a string.

Like PassageTokens, SectionTokens should have a `getText` member function, along with an appropriate constructor, but they should also have a `getType` member function, which returns the token's type (`LINK`, `GOTO`, `SET`, `IF`, `ELSEIF`, `ELSE`, `BLOCK`, or `TEXT`). These token types are described in more detail below. SectionTokens should also return an empty string when invalid.

3.3 Links

A link in the body of a passage will be denoted by double brackets: `[[` and `]]`.
Example link:

```
[[Example link]]
```

3.4 Commands

Commands are denoted by a single word and a colon immediately after an open parenthesis. Your IF interpreter should support 5 different commands, `(go-to:`, `(set:`, `(if:`, `(else-if:`, and `(else:`. In all cases, the SectionToken should begin at the open parenthesis and end at the corresponding close parenthesis. *Example commands:*

```
(go-to: &amp;quot;start&amp;quot;)
(set: $ateCake to true)
(if: $ateCake is true)
(else-if: $ateCookie is true)
(else:)
```

3.5 Blocks

Blocks are sections of a passage that follow if, else if, or else commands and denote what should happen when the given condition is true. Blocks will always start with an open bracket `[` and end with a close bracket `]`. As expected, the SectionToken should begin at the open bracket and end at the close bracket.

Note that an input file may have some white space between the if command and it's corresponding block—you should ignore this white space, as the block must be the next token after an if, else if, or else commend. Also, blocks can *only* follow these commands. If, for example, the first character of a passage was `[`, this character could be part of a link or text, but it cannot start a block.

One very important feature of blocks is that they may contain other features inside them, such as links, commands, text, or even other blocks. As a result, you need to be very careful when matching brackets so that you connect the bracket that starts a block to the correct bracket that ends it, not just the first end bracket that appears.

For this part of the project, you *do not* need to process anything inside of a block. A block should be a single SectionToken, regardless of what it contains. *Example block*

```
[All of that cake you ate earlier has made you [[drowsy]].]
```

3.6 Text

Text is a section of a passage containing ordinary text to be displayed. The corresponding SectionToken should start the character after the previous token ends or after the end of the `<tw-passagedata ...>` tag, if it is the first SectionToken in a passage, and it should continue until the first character of the next token or until the `</tw-passagedata>` tag, if it is the last token in the section.

4 Error handling

Your code will only be tested on valid input, so it does not need to be robust to reading errors in its input. A "real" tokenizer, though, should be able to recognize and report malformed input (e.g., an unterminated passage, command, or block) and handle or report them meaningfully.

5 Testing your code

You have been provided with a main function that will read in the data for an interactive fiction story and use your StoryTokenizer and PassageTokenizer classes to break down that story into its constituent tokens. You have also been provided with an example input you can use to test your tokenizers.

I would recommend that you implement and test StoryTokenizer and PassageToken before starting on PassageTokenizer and SectionToken. The most difficult part of the project will be implementing nextSection. I would start implementing this function by being able to detect links, ignoring everything else. After testing to make sure that works, add in the ability to detect the set command and debug. Then, add support for the other 4 commands, then detecting text in between the other tokens (treating blocks as ordinary text), and finally blocks, debugging after each major addition.

Important: By limiting the amount of code that you write between testing one version of your code and the next, you will drastically reduce the difficulty of debugging. In general, it is useful to think about the "minimum testable version" of your code when working on a large project, so that you do not try to debug the entire thing at the end.

6 Submission instructions and grading

You should submit header and source files for your StoryTokenizer, PassageToken, PassageTokenizer and SectionToken classes as a zip archive. You may combine all four into a single header and single source file, or you may submit four of each. If you do not combine the headers together, you should `#include` the other three headers at the top of your StoryTokenizer header (`storytokenizer.h`).

Your submission will be evaluated based on whether it compiles, as well as whether it can correctly tokenize inputs of increasing complexity when using the provided driver file.