Interactive Fiction Project, Part 2

1 Description

Over the course of the semester, you will develop an interpreter for interactive fiction. Interactive fiction (IF) is a text-based medium in which a reader is given passages of text interspersed with choices. In a sense, these stories resemble the "Choose Your Own Adventure" books 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 the passages that you tokenized in part 1 into their constituent components.

Like Part 1, this assignment is *individual*; however, you will be working with a partner in later parts of the project, so using good coding style is strongly encouraged. Also: the different parts of this project build on one another, so it is important that you do well on the earlier parts of the project, or you may need to spend time later fixing the bugs in earlier parts.

If your solution for Part 1 did not work correctly, please get in touch with a TA or the professor immediately—this project cannot be completed without a functioning StoryTokenizer.

2 Background information

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

3 Specifications

Your goal for this assignment is to write a pair of classes to tokenize the passages in interactive fiction stories. The "main" class, PassageTokenizer, will take in the text of a passage in an interactive fiction story (e.g., from the getText() method of PassageToken), and it will break this input up into PartToken objects, each of which represent an important section of the passage. Your code should be able to recognize a total of 8 different kinds of part 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 Part tokens

In order to correctly interpret each passage in a story, it's important to be able to separate out the different parts of that passage. Your PassageTokenizer class should implement `nextPart` and `hasNextPart`, which return the next PartToken and whether the passage contains another part, respectively. You should have a PassageTokenizer constructor that accepts a string.

Like PassageTokens, PartTokens 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. PartTokens should also return an empty string and type `TEXT` when invalid.

3.2 Links

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

Example link:

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

3.3 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 PartToken should begin at the open parenthesis and end at the corresponding close parenthesis.

Example commands:

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

3.4 Blocks

Blocks are parts 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 PartToken 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 *must* follow these commands, and they 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 PartToken, regardless of what it contains.

Example block

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

3.5 Text

Text is a part of a passage containing ordinary text to be displayed. Text tokens should contain all of the characters in the passage text that do not belong to a command, link, or block (including new lines and other white space). If the text in a passage contains HTML features (e.g., `<a>` tags or symbols like `&quot;`), you should just treat these as ordinary text; you don't need to do any HTML processing. Also, there should never be two text tokens in a row; you should combine all characters of uninterrupted text into a single token.

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.

5 Implementing your code

You have been provided with a main function that will read in a story from *input.txt* and use your PassageTokenizer and PartToken classes, along with your SectionTokenizer and PassageToken classes, to break down each passage of the story into its constituent parts. You have also been provided with an example input file you can use to test your tokenizer.

I recommend that you start implementing nextPart to only return tokens containing the links in the passage, ignoring everything else in the passage. After testing to make sure you can detect links, add in the ability to return tokens containing set commands (then debug). Afterwards, add support for the other 4 commands, then add text tokens in between the other tokens (treating blocks as ordinary text), and finally add support for blocks, debugging your code after each major addition.

Correctly identifying blocks that contain links or other blocks can be quite tricky. I recommend that when you try to process a block, you iterate through

the text character by character, counting how many “levels deep” in brackets you are. The first open bracket of the block puts you at level 1, every open bracket after that will put you one level deeper, every close bracket will reduce your level by 1, and the close bracket that brings you back to level 0 will be the one that matches the first open bracket.
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.

Do not wait until the last minute to start working on this part of the project; you will not finish in time. This part of the project is substantially more difficult than part 1.

6 Submission instructions and grading

You should submit header and source files for your StoryTokenizer, PassageToken, PassageTokenizer, and PartToken classes as a zip archive. You may split the headers and source for these classes into any number of files. If you do not combine the headers together, though, you should `#include` the other 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 passages of increasing complexity when using the provided driver file.