Interactive Fiction Project, Part 1

Due June 12, 2020

June 3, 2020

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 out the different passages in a work of interactive fiction.

Note: 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.

2 Background information

Reading in and interpreting text, a task often referred to as *parsing*, is a common problem in Computer Science. Often, the first step in parsing input is to *tokenize* the input. Tokenizing input involves breaking it down into smaller chunks (tokens), which can then be analyzed and annotated. Tokenization organizes the input clearly to make parsing or interpreting the input easier. For exceptionally complex input, this tokenization process may even be multi-level, with one tokenizer breaking the initial input into coarse tokens that are then fed into another tokenizer to be broken down into smaller tokens.

Specifications

Your goal for this assignment is to write a pair of classes to tokenize the passages in interactive fiction stories. The "main" class, StoryTokenizer, will take the text of an interactive fiction story as stored in an HTML file, and it will break this input up into PassageToken objects, each of which represent one passage in the IF story.

2.1 Passage tokens

Interactive fiction works are divided into passages, which are analogous to to a chapter in a novel. Passages are denoted in HTML using the `<tw-passagedata>` tag. Specifically, each passage will start with `<tw-passagedata ...>` and will end with `</tw-passagedata>`, and the body of the passage will be between these two tags.

In addition, the opening tag (`<tw-passagedata ...`) will specify some attributes of the passage. Out of the attributes given in the file, we are only interested in the `name` attribute, which is an important attribute for understanding how the passages in an IF story are connected together.

Example passage:

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

Your StoryTokenizer class should have two member functions: `hasNextPassage` and `nextPassage`. The `nextPassage` function should return PassageToken objects representing the passages in the story. The first time it's called, it should return a token with the first passage; the second time, a token for the second passage, and so forth.

The `hasNextPassage` function, meanwhile, should return whether the story has another passage; that is, should the code call `nextPassage` again, or have all of the passages already been returned? StoryTokenizer should also have a constructor that accepts a string containing the story to tokenize (i.e., the contents of the HTML file).

The PassageToken class is simpler. It should have two member functions, `getName` and `getText`, as well as an appropriate constructor. The `getName` member function should return the name of the passage, which appears as an attribute in the starting `<tw-passagedata>` tag. In the example passage above, `getName` should return “start”. The `getText` member function should return the body of the passage (all of the text between the starting tag `<tw-passagedata ...>` and the ending tag `</tw-passagedata>`). In the example above, this function would return “The body of the passage will be here.”, with newline characters before and after. An invalid PassageToken (e.g., the return result of `nextPassage` when there are no more passages) should return an empty string for its name and text. The arguments of the constructor and the data members of PassageTokens are up to you, as a PassageToken will only be constructed by the StoryTokenizer.

3 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., a passage with no name or an unterminated passage) and handle or report them meaningfully.

4 Implementing your code

You have been provided with a main function that will read in a story from `input.txt` and use your `StoryTokenizer` and `PassageToken` classes to break down that story into its constituent passages. Your tokenizer should appropriately ignore any text in the input file that is not part of a passage. You have also been provided with a couple of example input files you can use to test your tokenizer.

Though there is more than one way to implement your tokenizer, you may wish to take advantage of the `find`, `substr`, and/or `at` member functions of the `string` class when implementing your code. You can check the lecture slides or online documentation (`http://www.cplusplus.com`) to learn more about the arguments and return values of these member functions.

5 Submission instructions and grading

You should submit header and source files for your StoryTokenizer and PassageToken classes as a zip archive. You may submit one header file and one source file that defines both classes, or you may submit two of each. If you do not combine the headers together, you should `#include` the `PassageToken` header 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 when using the provided driver file.