CS 330: Homework 4
Bifur

Overview

Eliciting a reader's involvement can make a bad story okay and a good story great. (Adding involvement to an already great story sets the overflow bit.) The craftsmanship may be top-notch, the paints may be handground, and the composer may have died in its writing, but at the end of the day, the *influence* of a piece of art is our measure of its quality. In this homework, we will create a tool for writing and producing stories that maximize influence by drawing in readers through choices over the plot.

Consider this very simple story written in the Bifur language:

```
{
  title "Eg"
  page "Page 1"
}

It was a dark and stormy night... You debate {link "With Umbrella"}whether{end} or
{link "Without Umbrella"}not{end} to take your umbrella.

{page "With Umbrella"}
You leave with your umbrella. As soon as you step outside the wind takes hold of
your umbrella, but you maintain an iron grip.  You are carried to a distant land
and have a lot of adventures.

{page "Without Umbrella"}
You leave your umbrella behind. As you head out the door, however, it starts to
whimper. You {link "With Umbrella"}decide to take it{end}.
```

When processed by your interpreter, this source will be transformed into an HTML/JavaScript page. Initially, only Page 1's text will be visible. The reader will click on one of the two links to advance the story.

This homework is inspired by Twinery (`http://twinery.org`). Please skim through its description to get a feel for our task. Also view in a browser the HTML files in `<repo>/specs/grade_bifur` to see some example stories. For the simplest illustration of your task in this homework, check out `a_hemingway.in`. When this file is fed into your interpreter, a page like `a_hemingway.html` should be produced.

Requirements

Specifications do not tell you how to solve a problem—just what pieces may be used. The requirements described below and the descriptions above will need to be thought about and pieced together using your own good mind. You will likely need to read this document many times, ask questions, and afford yourself plenty of time to finish this homework.

Your solution is to meet the following specification:

- Place all files in directory `<YOUR-REPOSITORY>/bifur`.
- All code must run on `dplsubmit` and in an HTML5-compliant browser without extensions.
- Write your interpreter in `Interpreter.java`. Subclass `BifurParserBaseListener`, which is built by ANTLR based on your parser grammar. Its main method accepts a command-line parameter for the Bifur source file to parse. After lexing and parsing, it then walks the parse tree, triggering your callbacks. The HTML/JavaScript translation of the Bifur source is printed to STDOUT.
- Write `makefile` to build your interpreter. The default rule builds `Interpreter.class`. The `clean` rule removes files matching the following patterns: `*.tokens`, `*.class`, `*Lexer.java`, `*Parser.java`, and `*Listener.java`. The interpreter depends on the parser and lexer source, which depends on the two grammar files. Place your directory and the ANTLR Jar file in `../specs/grade_bifur` in the classpath.
- Have your interpreter terminate with non-zero exit status on the first lexer or parser error. The easiest way to do is to register an error handler that calls `System.exit`.
- Your terminals must be defined in a lexer grammar in `BifurLexer.g`, whose first line is:

  ```
  lexer grammar BifurLexer;
  ```

  Your lexer must define what's called an island grammar. Bifur source consists of islands of grammar-compliant code amidst oceans of plain, uninterpreted text. By default, consider the lexer to be in noncode mode. This default mode should only consider two lexemes: noncode sequences and opening curly braces. Noncode segments are sequences of 1 or more non-{ characters. When the lexer sees a {, it should enter ISLAND mode. The following production will switch modes when it encounters a curly brace:

  ```
  CODE_START : '{' -> mode(ISLAND);
  ```

  In ISLAND mode, one may encounter the following lexemes: double-quoted strings; keywords `if`, `else`, `end`, `page`, `link`; all standard arithmetic, relational, and logical operators; the assignment operator and its arithmetic friends (like `+=`); identifiers (which start with a lowercase letter and are followed by any letters or numbers); any number, positive or negative, fraction or whole; and a }. Whitespace is to be ignored. The following ANTLR code sections off your ISLAND mode productions and includes a production to switch back to the default mode when a closing curly brace is seen:

```
mode ISLAND;

CODE_END : '}' -> mode(DEFAULT_MODE);
```

- Your non-terminals must be defined in a parser grammar in `BifurParser.g`, whose first lines are:

  ```
  parser grammar BifurParser;
  options { tokenVocab=BifurLexer; }
  ```

  A Bifur story consists of one or more blurbs. A blurb is either code or a non-code sequence. Code is one or more statements sandwiched between braces. A statement is a conditional statement, an assignment statement, a page command, a title command, a link command, or an expression. An expression is a parenthesized expression; a bare identifier; a string; a number; or a multiplicative, additive, relational, or logical operation on subexpressions.

  An assignment statement consists of an identifier, an assignment operator (including `=`, `+=`, and its ilk), and an expression.

  A conditional statement consists of the keyword `if`, an expression, a then-block, an optional `else` and else-block, and the keyword `end`. Both blocks contain zero or more statements. However, writers may also exit ISLAND mode to write some plain text. If they close ISLAND mode inside the block, they must also reopen it to write a well-formed conditional statement (`end` is only significant if it appears in ISLAND mode). ISLAND mode may be exited and reentered any number of times before the conditional statement is completely expressed. PHP behaves similarly.

  Both page and title commands expect their respective keywords to be followed by a string parameter. A link command consists of the keyword `link`, a string, a block (just like the then- and else-blocks), and the keyword `end`.

- Based on your parser grammar, ANTLR will issue callbacks to your interpreter each time a production is applied. Through these callbacks, you generate a representation of a Bifur story in a language that we already know how to execute: HTML and JavaScript. This translation is the crux of all compiling and interpreting: rewriting our source in a language that a machine already knows how to run. Often the target language is machine code.

  Provided you support interaction like the examples, you have freedom in how you write your interpreter and implement your callbacks. I suggest you take the following overall approach:

  1. In your callbacks, print HTML and JavaScript code to construct the interactive page for a story.

2. Consider the main body of your HTML page to be a simple header and `div` for the page content:

   ```
   <h3 id="pagetitle"></h3>
   <div id="pagecontainer"></div>
   ```

3. Keep a pages dictionary/hash/map in JavaScript, binding a page's title to a function that returns its content. Pages are dynamically generated, so a page cannot just be stored as a string. What we store for each page is code that generates an HTML string.
4. Define a JavaScript `loadPage` function that when called loads the page with the title given as a parameter. It uses JavaScript's DOM to alter the header and `div`:

   ```
   document.getElementById("pagetitle").innerHTML = title;
   document.getElementById("pagecontainer").innerHTML = pages[title]();
   ```

5. Load the first page when the story is shown in the browser. When a reader clicks on a link, load the page with the given title.

More specifically, you might do the following things as productions are applied:

1. When the story is entered, start the HTML header and initialize any JavaScript code and data.
2. When the story is exited, close the HTML header, present the body, and load the first page.
3. When the title is exited, assign the title to `window.document.title`.
4. When a page command is exited, start a new entry in the pages hash using the given title by beginning the definition of the page's function. The function will dynamically assemble and ultimately return the text of the page. You may need to finish off any previously started page.
5. When a non-code blurb is exited, concatenate its text to the current page's text.
6. When a conditional statement is entered, issue the beginning of a JavaScript `if` statement to the current page's function. Since Bifur conditional expressions are a subset of JavaScript's, you can use the text of the Bifur condition directly as JavaScript source.
7. When a conditional statement is exited, close off the `if` statement previously started.
8. When an else block is entered, close off the then-block and start the else block.
9. When an expression statement is exited, arrange to have the value of the expression added to the page's text when the function is called. Since Bifur expressions are a subset of JavaScript's, we can use the Bifur expression directly as JavaScript source.

10. When an assignment statement is exited, add the statement directly to the code of the current page. Since Bifur expressions are a subset of JavaScript's, we can use the Bifur expression directly as JavaScript source.
11. When a link statement is entered, issue the start of an anchor tag to the page's text:

    ```
    <a href="#" title="<TITLE>"
       onclick="loadPage(\'<TITLE>\'); return false;">
    ```

    Replace `<TITLE>` with the page's title.
12. When a link statement is exited, close the anchor tag previously started.

Occasionally a page's content or title may contain quotation marks or character sequences that need to be represented by HTML entities. Also, the string lexemes matched by the lexer will be double-quoted. You likely will need to sanitize your strings by removing their surrounding quotation marks and replacing internal double-quotation marks with `&quot;`, single-quotation marks with `&#39;`, and linebreaks with `<br/>`.

- Exit on malformed Bifur source. ANTLR provides default error messages which are sufficient for this introduction to language translation. There are likely legal Bifur expressions that are not legal JavaScript. You are welcome to report those on Piazza, but you need not exceed the grammar specified above.

- Write a Bifur story at least 10 pages in length and share its source and HTML on Piazza in a post tagged `bifur`. Your thoughtful contribution on open-ended tasks like this provide great material for reference letters.

Submission

This homework is graded by hand and with help from the grading script, which can run from your homework directly with `../specs/grade`. Your assignment is expected to fully meet the requirements above, those checked by the grading script, and the following:

1. Variable names should be meaningful and accurate.

2. Non-obvious parts of your code should be commented.

3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. (If you find violators of this rule, please let me know.) Write your own code.

The grading script sends your instructor an email when it successfully completes. All that remains is for you to push your code to Bitbucket.