CS246E—Assignment 3 (Fall 2019)

B. Lushman

Due Date 1: Thursday, November 7, 5pm
Due Date 2: Thursday, November 14, 5pm

Part of this assignment is due on Due Date 1; the rest is due on Due Date 2. See the assignment questions for specifics. All code is to be written in C++14.

1. In this problem and the next four, we will revisit the `egrep` tool that you wrote as part of A1. The scope of that problem was limited, in that we permitted at most one regular expression operator in the search pattern. We now aim to solve the general problem.

   When it comes to processing general regular expressions, one of the first questions we must confront is that of *precedence*: does `c*d|a` mean `c*(d|a)` or `(c*d)|a`? We will assign the highest precedence to `*`, then concatenation, then `|`. For example, in the case of concatenation over `|`, we would interpret the regular expression `(a|b)(c|d)|e` as meaning `((a|b)(c|d))|e`.

   We can express these precedence relationships in the form of a "data definition" for regular expressions. A regular expression is:

   - a disjunction of two regular expressions, or a regular expression without disjunctions;
   - where a regular expression without disjunctions is a concatenation of two regular expressions without disjunctions, or a regular expression without disjunctions or concatenations;
   - where a regular expression without disjunctions or concatenations is a star operator applied to a regular expression without disjunctions or concatenations, or a regular expression without disjunctions, concatenations, or stars;
   - where a regular expression withoud disjunctions, concatenations, or stars is a single word (which may be empty), or a pair of parentheses surrounding a regular expression.

   The above description is what is known as a *concrete syntax*: it leaves no question about the precedence of operators, and leaves only one way for a given regular expression to satisfy the definition (modulo associativity of the disjunction and concatenation operators).

   But if already know how the operators of a given regular expression are to be grouped, we may use a much simpler *abstract syntax* — a regular expression is:

   - a disjunction of two regular expressions;
   - or a concatenation of two regular expressions;
   - or a star operator applied to a regular expression;
   - or a single word.

   For this problem, you are to create classes corresponding to these four types of regular expression: `Disjunction`, `Concatenation`, `Star`, `Word`; with an abstract `RegExp` superclass from which they may inherit. For example, the regular expression `(cat|dog)*bird` could be encoded as

```
new Concatenation{
  new Star{
    new Disjunction{
      new Word{"cat"},
      new Word {"dog"}
    }
  },
  new Word{"bird"}
};
```

Your classes must provide a virtual method called `matches` that takes a `string` and answers `true` if the string exactly satisfies the regular expression (i.e., with no characters left over), and `false` otherwise. You may assume that the word in the `Word` class contains no special characters, or rather, if it does, that those characters should be taken literally.

We will provide a sample main program that contains this example, but you must augment this with your own testing. As always, expect your solutions to be hand-marked.

The files you submit will at least contain the file `re.h`. That is the file that our testing will include. You may use this file to include the headers that actually make up your solution. Also don't forget to submit the accompanying `.cc` files.

Provided test harness: `q1main.cc`

No test suite required; submit your code on Due Date 2.

Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

2. Rewrite your solution for Question 1, such that it uses `std::unique_ptr` instead of raw pointers. The example from Question 1, rephrased as follows, should work with your implementation:

```
make_unique<Concatenation>(
  make_unique<Star>(
    make_unique<Disjunction>(
      make_unique<Word>("cat"),
      make_unique<Word>("dog")
    )
  ),
  make_unique<Word>("bird")
);
```

The files you submit will at least contain the file `re.h`. That is the file that our testing will include. You may use this file to include the headers that actually make up your solution. Also don't forget to submit the accompanying `.cc` files.

Provided test harness: `q2main.cc` **(do not submit)**

No test suite required; submit your code on Due Date 2.

Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

3. The last challenge in creating a general-purpose regular-expression matcher is parsing. Specifically, we wish to take a string `s`, containing a regular expression in human-readable format, and produce an equivalent `RegExp` object.

   There are many parsing algorithms; one of the simplest is known as *recursive descent*. A recursive descent parser works by having one parsing function for each element of abstract syntax, i.e., for each kind of expression. Each of these functions consumes a `string` parameter and produces a parsed expression object:

   ```
   std::unique_ptr<Item> parseItem(const string &s); // Whatever type Item happens to be
   ```

   For this problem, we recommend having one such function for each of `Disjunction`, `Concatenation`, `Star`, and `Word`. Each such function would first determine whether the string represents the kind of expression that the function is looking for. If so, then the function splits the string into two (or one, in the case of `Star`), and (mutually) recursively calls the appropriate recognition function for each part; it then creates the appropriate regular expression object with the parsed parts as subobjects. If not, it calls the recognition function with the next lowest precedence, which will attempt to parse the string as that kind of expression.

   For example, the `parseDisjunction` function should work roughly as follows:

   - determine if `s` contains a `|` operator, not nested inside parentheses;
   - if so, find the first such operator, and split the string into the part before the `|`, and the part after the `|`; parse the two pieces, and then build a `Disjunction` object from the results;
   - if not, parse `s` as a concatenation (i.e., call `parseConcatenation`)

   To facilitate testing of your parser, add a virtual `print` method to your `RegExp` class hierarchy. This virtual method should take an lvalue reference to an `ostream` as parameter, and return the same reference as its result. The sample executable will show you how the different kind of expressions should be printed. Also include the following overload of `operator<<` in your solution:

   ```
   std::ostream &operator<<(std::ostream &out, RegExp &re) {
     return re.print(out);
   }
   ```

   We will provide a test harness, suitable for testing your code. After each command you issue to the test harness, it will print the parsed expression, using the method described above.

   Note: In the interest of simplicity, you will treat each multi-character word as a concatenation of single-character words. So the regular expression `dog` will print as

   ```
   Concatenation (d,Concatenation (o,g))
   ```

   You will need to submit a test suite for your program, in a format compatible with `runSuite`.

   The files you submit will at least contain the file `re.h`. That is the file that our testing will include. You may use this file to include the headers that actually make up your solution. Also don't forget to submit the accompanying `.cc` files.

   Provided test harness: `q3main.cc` (do not submit)

   Submit your test suite on Due Date 1; submit your code on Due Date 2.

   Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You

must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

4. Now that you have a working regular expression recognizer, the next step is to look for a match for a regular expression *within* a string. Write the following function:

   ```
   bool containsMatch(RegExp *re, const string &s);
   ```

   It should answer `true` if `re` matches any substring of `s`. Do not worry if the check takes a quadratic number of attempts in the length of `s`. There are more efficient ways to solve this problem, but those are the realm of CS 241.

   Once you have this working, determine how to add support for the special words `^` and `$`. In order to be compatible with our testing, these must be handled by creating two additional classes called `Begin` and `End`, whose constructors take no arguments. To simplify this process, you are permitted to assume that the input string is coded in 7-bit ASCII, and may use the characters with ASCII values 128 and 129 as indicating the beginning and end of the line, respectively:

   ```
   bool containsMatch(RegExp *re, const string &s) {
     string s2 = '\200' + s + '\201';
     // then proceed using s2
   }
   ```

   Once you have this working, add support for the dot (`.`) operator, via a class called `Dot`.

   The files you submit will at least contain the file `re.h`. That is the file that our testing will include. You may use this file to include the headers that actually make up your solution. Also don't forget to submit the accompanying `.cc` files.

   Provided test harness: `q4main.cc` (do not submit)

   Submit your test suite on Due Date 1; submit your code on Due Date 2.

   Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

5. Finally, complete your `egrep` program. For this problem, you are not required to support the command-line flags that were expected of you on A1. Your program should be invoked as follows:

   ```
   my-egrep pattern file file ...
   ```

   It will print all lines in each `file` that contain a match to `pattern`. If no files are given, it takes its input from stdin.

   The `pattern` will be a regular expression involving the operations you were asked to support. You should parse the regular expression using your solution from problem 3, and then proceed to print lines from the files that contain a match to the pattern.

   If you have solved all of the problems before this one, this should be a straightforward exercise.

   Provided test harness: none. Submit your complete program, including a Makefile that builds the target `a3q5`.

   No test suite required; submit your code on Due Date 2.

Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.