Programming Comps — January 13-16, 2009

1 Background

You have been working in the exciting field of network protocol design for years and have become dissatisfied with the code quality of protocol implementations. Together with your colleague, Ben Bitdiddle, you devise a set of tools to test code fragments against each other to ensure that the structure of the messages they emit and accept are identical.

Messages consist of *atoms*, which will be represented as capital letters. `ACCD` is a message composed of three different types of atoms, the second type is repeated twice. Ben has written a tool for your code base to find the messages that can be emitted or accepted by a code fragment (assume that no single fragment accepts *and* emits fragments — only one or the other). The analysis reports the messages that may be emitted or accepted by a program fragment in a compact form called a *protocol expression*.

In a protocol expression, a capital letter represents an individual atom. Consecutive expressions indicate that the first expression is emitted/accepted, followed by the second. Parentheses indicate a code loop and wrap a protocol expression that may emit/accept zero or more times. A matched pair of curly brackets surrounding protocol expressions that are separated by vertical bars represents a branch or switch. Exactly one of the bar separated protocol expressions will emit/accept. Finally, an empty string is a protocol expression for a fragment that emits nothing. No other characters may appear in a protocol expression. Some examples:

$ABBC$ – Emits only the message `ABBC`

$(AB)X$ – May emit `X`, or any number of repeated `AB`s followed by `X`, such as `ABABABX`.

$B\{C|D|E\}$ – Emits `BC`, `BD` or `BE`.

$(D\{B|\})$ – Emits the empty message or a mix of `D`'s and `B`'s that begins with `D` and has no consecutive `B`'s.

2 Tool Descriptions

Using the output from Ben's analysis, you will build tools to improve protocol implementation quality. These tools will check protocol expressions against messages and protocol expressions against each other.

As you may have already realized, protocol expressions are a lot like regular expressions. You will use your knowledge of regular languages, DFAs, and NFAs to implement your tools. It is perfectly legal (and recommended) that you refresh your knowledge from non-human sources such as textbooks and webpages.

2.1 Matcher

Your first tool must answer questions of the form, "Could *expr* have emitted *msg*?" It will take two inputs, a protocol expression and a message. Print the line[1] "<msg> can be emitted by <expr>." or "<msg> can not be emitted by <expr>." as appropriate.

2.2 Verifier

Your second tool is used to compare two protocol expressions. For example, you may want to check that a client and server emit and accept the same set of messages. The verifier will again take two inputs, but both will be protocol expressions. You should think of the first expression as the message emitter, and the second as the acceptor.

If the set of messages emitted/accepted by both expressions is the same, print the line "<expr1> and <expr2> are verified.".

[1] A line is a sequence of characters terminated by a newline.

If the first expression can emit messages that the second can not, you will generate an example of such a message and print the line "<msg> can be emitted by <expr1>, but not accepted by <expr2>.".

If the first expression cannot emit a message that the second expression accepts, generate an example of such a message and print the line "<msg> can be accepted by <expr2>, but not emitted by <expr1>.".

Your tool must generate an example in each direction (print two lines, in the given order) if both exist.

This is a programming exam, not an algorithms exam, so we ask that you implement `verify` in the following way: Convert the protocol expressions to NFAs, then convert the NFAs to DFAs. Combine the DFAs to produce a single DFA that will let you determine if the DFAs match, and find counter-examples if they do not.

This task is NP-Complete! Be sure to explain how that fact impacts your design decisions, and any limitations it places on your tools.

3 Your Task

You will implement each of these tools. Your implementation must meet the following specifications.

- Hand in **one** file named <username>-comps.tar.gz contains (1) all of your source code, (2) a script `build` that will compile it on a department Linux machine, and (3) a file named Protocols.pdf that describes your work *and* includes your code. Protocols.pdf should describe your implementations in detail, including a justification of your language and data structure choices, structure of your code, and your testing procedures. For grading convenience, your code must be included in Protocols.pdf as an appendix. Hand in by emailing the tar.gz file to `cpap` by 4pm Friday.
- Your tools should be very careful about input and output. They should never crash on *any* input. The more gracefully they report errors, the better. We have supplied a test case for each tool in `~jj/protocols`. The following commands should produce *no output*. The tests are particularly instructive for handling empty messages and expressions.

  ```
  ./match ~jj/protocols/match-01.in | diff - ~jj/protocols/match-01.out
  ./verify ~jj/protocols/verify-01.in | diff - ~jj/protocols/verify-01.out
  ```

- After `./build` is run from your unpacked files, there should be two programs in the current directory, `match` and `verify`.[2] Each program should be runnable in two ways. If given two command line arguments, they will execute as described above on those arguments. If they are given *one* command line argument, the argument is a file containing a pair of arguments per line, separated by a space. Match or verify them sequentially.
- You may not use existing regexp, DFA, NFA, or graph libraries.
- All work must be your own. This includes code *and* the thinking behind it. You may not discuss the problem *in any way* with anyone else. Your only conduit for clarifications is by email to `jj`. Responses to such questions will be cc'd to the `comps` mailing list.

4 Evaluation Criteria

Your implementation will be evaluated based on **all** of the following criteria, some objective, and some subjective.

- Structure, readability, and maintainability of your code.
- Correctness, speed, and memory efficiency.

It is important that your work passes a critical threshold with respect to all criteria. If you cannot complete all specifications, focus on the main algorithm and functionality that is required. Even the most efficient code will be regarded a failure if it is poorly documented, barely maintainable, or error prone. On the other hand, a software engineering masterpiece that does not work as specified is equally unacceptable.

Good luck!

[2] They might be wrapper scripts to invoke your real program(s).