Written Set 1: Lexical Analysis

In this first written assignment, you'll get the chance to play around with the various constructions that come up when doing lexical analysis. This problem set will start off by reviewing those constructions, then ask you to think critically about them. You'll be considering different algorithms for lexical analysis, along with some of the limitations of what we presented in class. In particular, you'll explore the various algorithms for generating automaton-driven scanners and for resolving conflicts that might arise in them.

This assignment is due on Friday, July 6. I suggest starting early while the material is still fresh in your mind. Feel free to email us with questions or to drop by office hours.

Overall, this assignment is worth 10% of your total grade in the course. Each problem is weighted equally.

Collaboration on problem sets

Although you can discuss ideas with others, you must submit your own independent solution and properly give credit to anyone you worked with.

Submission instructions

There are two ways to submit this assignment:

1. Submit a physical copy of your answers in the filing cabinet in the open space near the handout hangout in the Gates building. If you haven't been there before, it's right inside the entrance labeled "Stanford Engineering Venture Fund Laboratories."
2. Send an email with an electronic copy of your answers to the staff list at cs143-sum1112-staff@lists.stanford.edu. Please include the string [WA1] somewhere in the subject line so that it's easier for us to find your submission.

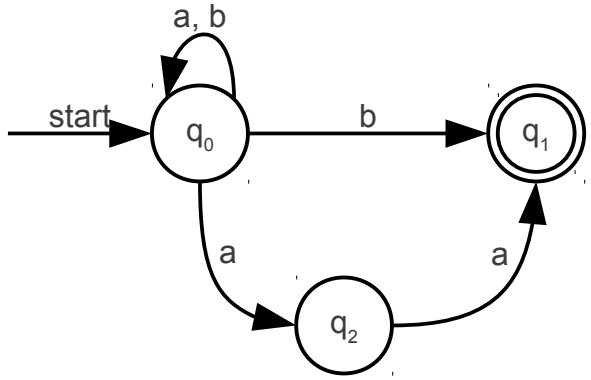
Good luck!

Due: Friday, July 6th at 5 p.m.

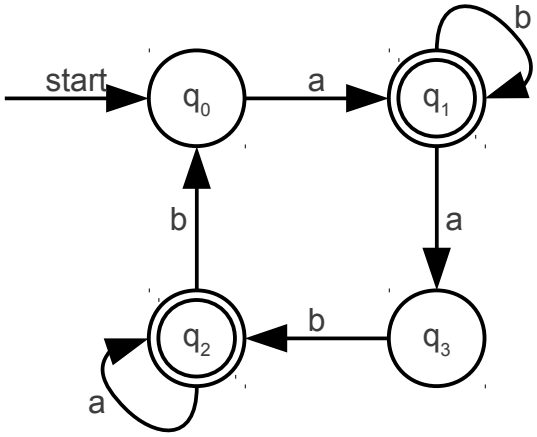
Problem One: Subset Construction

Use the subset construction to convert the following NFAs into DFAs. You can assume that the alphabet is just the two letters **a** and **b**. Remember that in a DFA, every state must have a transition defined on every symbol.

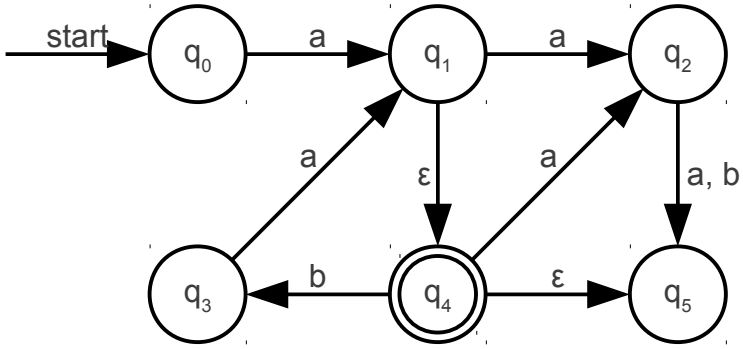
i.



ii.



iii.



Problem Two: Maximal Munch

Given the following tokens and their associated regular expressions, show what output is produced when this **`flex`** scanner is run over the following strings:*

```
%%
a*b             printf("1");
(a|b)*b         printf("2");
c*              printf("3");
```

i. aaabccabbb

ii. cbbbbac

iii. cbabc

Problem Three: The Limits of Conflict Resolution

In class we discussed one particular means for resolving conflicts that occur when using regular expressions to scan an input. Our approach was to use maximal-munch to always choose the longest possible match at any point, then to break ties based on the priorities of the regular expressions. However, this is not the only way that we could have resolved conflicts. Recall from lecture our example with two types of regular expressions: an expression for the keyword **`for`** and an expression for identifiers:

```
%%
"for"                       { return T_For; }
[A-Za-z_][A-Za-z0-9_]*      { return T_Identifier; }
```

We saw how the string **fort** could be tokenized in nine possible ways, based on how we chose to apply the regular expression. In this case, maximal-munch algorithm dictates that we would scan the string as the identifier **fort**.

However, in some cases we may have a set of regular expressions for which it is possible to tokenize a particular input string, but for which the maximal-munch algorithm will not be able to break the input into tokens. Give an example of such a set of regular expressions and an input string such that

- The string can be broken apart into substrings, where each substring matches one of the regular expressions, but
- The maximal-munch algorithm will fail to break the string apart in a way where each piece matches one of the regular expressions.

Additionally, explain how the string can be tokenized and why maximal-munch will fail to tokenize it.

---

* If you want to, you can actually use **`flex`** to compile and run this scanner to check your answer. However, please be sure that you understand why the output is as it is.

Problem Four: Converting Extended Regular Expressions

In Wednesday's class, we discussed a recursive algorithm for converting regular expressions into NFAs. If you'll recall, we associated with each regular expression an NFA with three properties:

1. The generated NFA has exactly one terminal state.
2. The generated NFA has no transitions into its start state.
3. The generated NFA has no transitions out of its terminal state.

Additionally, we encountered three shorthand notations that extended our standard regular expression grammar:

1. **R?**, which matches zero or one copies of **R**,
2. **R+**, which matches one or more copies of **R**, and
3. **R{n}**, which matches exactly *n* copies of **R**.

Show how to extend the existing RE-to-NFA construction to support these new regular expressions. In particular, you should show the automata you would construct to match each of these expressions. Briefly explain why each construction is correct.

Problem Five: Right-to-Left Scanning

The scanning algorithm we covered in lecture is based on a left-to-right scan of the input. We convert the series of regular expressions describing the lexical analysis into a matching automaton, then feed the characters from the input file one-at-a-time into the automaton from left-to-right.

However, we just as easily could have considered doing a *right-to-left* scan of the input, in which we feed the the *last* character into the automaton, then the second-to-last character, etc. Notice that this means that the matching automaton we create must match a given pattern *in reverse*, since it will be seeing the characters in reverse order.

i. Modify the existing algorithm for converting regular expressions to NFAs so that the generated NFA accepts the **reverse** of strings that match the regular expression. Briefly justify why your construction is correct.

ii. Give an example of a set of regular expressions and a string so that the left-to-right scan of the string produces a different set of tokens than the right-to-left scan. Assume that you're using the maximal-munch algorithm for conflict resolution.

Problem Six: Slowing Down `flex` Scanners

As mentioned in class, in order to implement maximal-munch scanning, we run a matching automaton over the input, keeping track of the last location of a match. When the automaton gets stuck, the scanner reports the last known munch, the restarts the scanner just after the reported token. Because the scanner may have to back up over many characters that it has already scanned, it is possible for the scanner generated from a set of regular expressions to reread parts of the input string many times, causing the scanner to run more slowly.

As an example, consider the following **`flex`** script:

```
%%
"photo"              { printf("PHOTO"); }
"photosynthesis"     { printf("PHOTOSYNTHESIS"); }
.|\n                 { printf("x"); }
```

If we run this script on the input **photosynthesize**, the scanner will note that a match exists with the prefix **photo**. It will then continue reading up until it encounters the **z**, at which point it will note that none of the above three regular expressions could match. The scanner will then match **photo**, outputting the text **PHOTO**, and will restart the scanner at the **s** just after **photo** in the input. It will then output an **x** for each of the remaining letters, so the overall output will be **PHOTOxxxxxxxxx**.

In this example, the scanner had to back up over half the characters in the input. Consequently, the scanner ended up revisiting all of the characters in the second half of the input twice. This is noticeably slower than a single pass over the input.

However, it's possible to come up with a set of regular expressions so that on certain inputs, the scanner ends up doing substantially worse than this. In fact, there are regular expressions that on certain strings of length n end up spending $\Theta(n^2)$ time rereading the characters in the string.[*]

Your job is to pick

1. A set of regular expressions, and
2. A function f that accepts an integer n and produces a string of length n

such that a maximal-munch scanning algorithm for those regular expressions, when run on a string $f(n)$, spends $\Theta(n^2)$ time scanning. Again, you have full control of what regular expressions are used and what strings they're run on. The point of this question is to let you think about the pathological cases of maximal-munch scanning by seeing how the runtime scales quadratically in some cases. If it will make it easier, you can have your function only work for n larger than an integer of your choice.

You do not need to formally prove that your construction is correct, but you should offer a justification as to why the runtime is $\Theta(n^2)$.

---

[*] $\Theta(n^2)$ is similar to $O(n^2)$, except that it implies a *tight* bound instead of an *upper* bound. If a function is $O(n^2)$, it could also be $O(n)$. However, if a function is $\Theta(n^2)$, it grows asymptotically at the rate of n^2.