CS 6410: Compilers

Fall 2019

HW 1 – Regular Expressions and Scanners
Solutions

Assigned: Thursday, September 27, 2023, Due: Saturday, October 14, 2023

Instructor: Tamara Bonaci
Khoury College of Computer Sciences
Northeastern University – Seattle

Submission Guidelines

- Please submit your homework as a single .pdf file through Canvas.
- You do not have to type in your submission - hand-written and then scanned, or photographed documents are fine, as long as your document is readable.
- This assignment is meant to be worked on individually, and you should submit it by **11:59pm on Saturday, October 13, 2023**.

Hints on Regular Expressions

For questions that ask for regular expressions, you **must** restrict yourself to the basic operations and abbreviations, including:

- Concatenation,
- Alternation, |
- Kleene closure, $*$
- Positive closure, $+$,
- Question mark, ?, and
- Character classes, [. . .]

Additionally, you may specify character classes containing all characters except for specific characters, e.g., $[\wedge abc]$. You can also use abbreviations, and be somewhat informal when the meaning is clear, e.g., allexceptx $= a|b|c|...|w|y|z$. You should not use additional "regular expression" operations found in libraries or languages like perl, python, or ruby, or in unix tools like grep and sed. In particular, you cannot use $not(re)$ as a regular expression that matches all other regular expressions except for re.

Problem 1

For each of the following regular expressions, please give:

1. An example of two strings that can be generated by the given regular expression,
2. An example of two strings that use the same alphabet, but cannot be generated, and
3. An English description of the set of strings generated (for example, "all strings consisting of the word 'cow' followed by 1 or more 'x's and 'o's in any order", not just a transliteration of the regular expression operations into English)

Regular expressions are as follows:

- $(\text{to}(\text{m}|\text{y}))^+$
- $\text{h}(\text{i}|\text{o})(\text{pity})^*$
- $((\epsilon|206)299)^?$

Solution:

1. Some example strings that can be generated by the given regular expressions include:
   - **(to(m|y))**$^+$: tom and toy
   - **h(i|o)(pity)***: hi and hopity
   - $((\epsilon|206)299)^?$: 299 and 206 299
2. Some example strings that use the same alphabet, but could not have been generated by the given regular expressions include:
   - **(to(m|y))**$^+$: to and toto
   - **h(i|o)(pity)***: pity and hpity
   - $((\epsilon|206)299)^?$: 206 and 299 299
3. The exact transliterations of the given regular expressions are as follows (*please not that we did not ask you for the exact transliterations*):
   - All strings consists of one or more repetitions of substring `to`, followed by an `m` or a `y`.
   - All strings consisting of a letter `h`, followed by a letter `i` or a letter `o`, followed by zero or more repetitions of a substring `pity`.
   - Zero or one repetition of either an empty character (ϵ), or `206`, followed by `299`.

   We actually asked you for an English language description of the given regular expressions. More specifically, we asked you for something like this:
   - **(to(m|y))**$^+$: This RE can, for example, generate the strings `tom`, `toy`, `tom tom`, `toy toy`. It cannot generate strings containing letters other than `t, o, m, y`, strings not starting with `to`, or strings where only letters `m` or `y` repeat multiple times, such as strings `tommmmm`, `toyyyyy`.
   - **h(i|o)(pity)***: This RE generates strings consisting of letters `h, i, o, p, t, y`, and it cannot generated strings containing any other letters. It generates strings where every letter `h` is immediately followed by either letter `i`, or letter `o`, and those two letters can be followed by zero or more repetitions of a substring `pity`. That means that strings `hi, ho` are allowed, but strings that don't start on `h` are not, for example `ipity, opity, pity, pity pity`.
   - $((\epsilon|206)299)^?$: This RE generates strings that can start either with an empty character, or with a substring `206`. The allowed string have to finish on substring `299`, and the whole sequence can be repeated at most one time.

Problem 2

Please state regular expressions that generate the following sets of strings:

- All strings of d's and e's with at least 3 d's.
- All strings of d's and e's where e's only appear in sequences whose length is a multiple of 2 (a few examples: deed, eeeedeeddd, d and ϵ are in this set; ded, e, deded, and deede are not).
- All strings of lower-case letters that contain the 5 vowels (aeiou) exactly once and in that order, with all other possible sequences of lower-case letters before, after, or in between the individual vowels.

Solution:

- The following regular expression generates strings with at least three `d`s, from the alphabet `d, e`:
  `(d|e)*d(d|e)*d(d|e)*d(d|e)*`
  Note the use of the very common form of interspersing the required characters with complete wildcards (which should give some hint as to the popularity of globbing in file-manipulation, as opposed to full-strength regular expressions; this is equivalent to the globbing `*d*d*d*`, except that we explicitly limit the alphabet to '`d`' and '`e`').
- The following regular expression generates strings where '`e`'s appears only in even-length runs, from the same alphabet as in the previous question:
  `(d|ee)*`
  This regular expression also generates an empty string, which was implied by the problem.
- The following regular expression generates strings that contain each of the lowercase vowels, exactly once, and in alphabetical order:
  `[~aeiou]*a [~aeiou]*e [~aeiou]*i [~aeiou]*o [~aeiou]*u [~aeiou]*`
  Please note that we've used the symbol ∼ here to indicate negation, meaning anything but `[a, e, i, o, u]` since the typesetting system used for the solutions makes special use of the caret character.

Problem 3 (Cooper and Torczon, Problem 2.2.1)

Please informally describe the languages accepted by the finite automata (FAs) depicted in Figure 1.

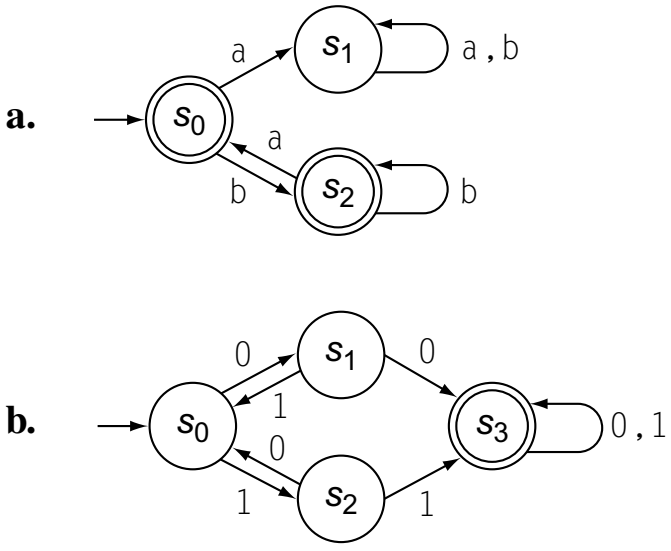

Fig. 1. Finite automate used in Problem 3.

Solution:

1. The given DFA depicts a regular expression that allows us to generate strings that start with character `a`, followed by zero or more repetitions of characters `[a, b]` (the upper branch of the automaton). It also allows us to generate strings that start with one or more repetitions of character `b`, which may be followed by one character `a`, followed by zero or more repetitions of of characters `[a, b]`. The regular expression can be formally described as $\texttt{b}^{\star}\texttt{a}^{?}\texttt{(a|b)}^{\star}$.
2. The given DFA allows us to generate strings that start with exactly two zeros or exactly two ones, followed by zero or more repetitions of characters `[0, 1]`. It also allows us to generates strings that start with zero or more alternations between 0 and 1 or 1 and 0, followed by two consecutive zeros or two consecutive ones, followed by zero or more repetitions of characters `[0 ,1]`. The regular expression can formally be described as `((0 1 | 1 0)* (0 0 | 1 1) (0, 1)*`.

Problem 4 (Cooper and Torczon, Problem 2.2.2)

Please construct a DFA accepting each of the following languages:

1. {w in {a,b} * | w starts with 'a' and contains 'baba' as a substring}
2. {w in {0,1}* | w contains '111' as a substring and does not contain '00' as a substring}

Solution: The resulting DFAs are depicted in Figures 2 and 3.

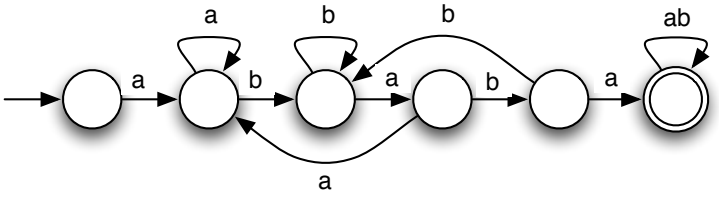

Fig. 2. Deterministic finite automate generated in Problem 4 (a).

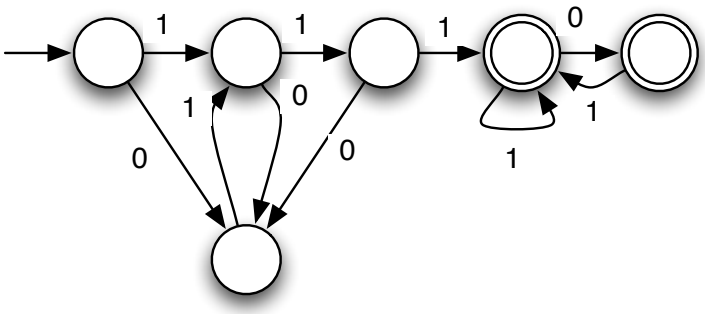

Fig. 3. Deterministic finite automate generated in Problem 4 (b).

Problem 5

In The C Programming Language (Kernighan and Ritchie), an integer constant is defined as follows:
An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. Octal constants do not contain the digits 8 or 9. A sequence of digits preceded by $0x$ *or* $0X$ *(digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. An integer constant may be suffixed with the letter u or U, to specify that it is unsigned. It may also be suffixed by the letter l or L to specify that it is long.*

1. Write a regular expression that generate C integer constants as described above.
   Hint 1: you may want to break the solution down into a regular expression with several named parts, if it makes things easier to write and read – which it probably will.
2. Draw a DFA that recognizes integer constants as defined by your solution to part (a). You may draw this directly; you don't need to formally trace through an algorithm for converting a regular expression to a NFA, and then constructing a DFA from that. However, you might find it useful to do so at least partially.
   Hint 2: You might find it helpful to alternate between designing the DFA and writing the regular expressions as you work on your solution.

Solution: The most common issues in this problem are handling the optional '`u`' and '`l`' at the end. The English description is a bit hard to read, but it does allow either one, or both, or neither. It also allows either to be capitalized or lowercase, as well as for them to appear in either order. A solution with (`u|U|l|L`)?, handles the case issue, but allows only one or the other, and not both. A solution like (`u|U|l|L`)* does generate all valid combinations, but it also includes things like "`uu`". Interestingly, a number of DFAs implemented a correct solution, even though regexps did not (the correct DFA is simpler to construct than the correct regexp).

1. One way to solve the problem is to supply a "set of regular expressions" with one for each kind of constant. That is a fairly easy way to think about the problem, and is one that you might use in a scanner specification to separate out decimal, octal, and hex constants for processing. But any answer that generated all (and only) the required set of strings was fine.
   We assume that the alphabet and sort order are such that [`0` − `9`] is exactly the digits '`0`', '`1`', '`2`', '`3`', '`4`', '`5`', '`6`', '`7`', '`8`', and '`9`'; [`0` − `7`] is exactly the digits except '`8`' and '`9`'; and that [`a` − `fA` − `F`] is exactly the letters '`a`', '`b`', '`c`', '`d`', '`e`', '`f`', '`A`', '`B`', '`C`', '`D`', '`E`', and '`F`'. All of these things are true for ASCII and other widely used character sets these days.
   We use the $a?$ operator to mean $(a|\epsilon)$.
   - (decimal integers) (`0`|([`1` − `9`][`0` − `9`]*)) (([`lL`]?[`uU`]?)|([`uU`]?[`lL`]?)) Here we have two cases for the number part: a decimal constant may only start with a '0' if it *is* 0, otherwise it must start with a non-zero digit (or else it would be an octal constant, which gets picked up below).
   - (octal integers) `0`[`0` − `7`]$^+$ (([`lL`]?[`uU`]?)|([`uU`]?[`lL`]?)) The number "`0`" is included as a decimal constant above and is not generated by this regular expression.
   - (hexadecimal integers) `0`(`x`|`X`)[`0` − `9a` − `fA` − `F`]$^+$ (([`lL`]?[`uU`]?)|([`uU`]?[`lL`]?))

   If we want to combine these into a single regular expression, we can factor out the 'ul' part, and get:
   (`0`|([`1` − `9`][`0` − `9`]*)|(`0`[`0` − `7`]$^+$)|(`0`(`x`|`X`)[`0` − `9a` − `fA` − `F`]$^+$)) (([`lL`]?[`uU`]?)|([`uU`]?[`lL`]?))
2. (Solution omitted)

Problem 6

In C programming language, a comment is a sequence of characters between characters $/*...*/$. Write a set of regular expressions that generate C-style comments. You can restrict the alphabet to lower-case letters, digits, spaces, newlines ($\backslash n$), carriage returns ($\backslash r$) and the characters $*$ and $/$. Also, remember that in C comments do not nest (i.e., a $*/$ marks the end of a comment no matter how many times $/*$ appears before it.)

Hint 3: be careful about what is included in the ... between /∗ and ∗/.

Solution: Some things to keep in mind about C-style comments:

- It takes both the '*' and the '/' to start or stop; either character alone, or with anything else between the pair, doesn't do it.
- a second '/*' inside a comment is part of the comment, and doesn't attempt any sort of nesting.
- Fonts are used carefully below: ∗ is the literal asterisk character, and * is the regular expression Kleene closure operator.

One regular expression that generates C-style comments is

$$/*([\sim*]^*|(*^+[\sim/*]))^**^+/$$

Some things to note about the design of this regular expression are:

- This takes advantage of the negation operator, so that we don't have to specify the rest of the alphabet.
- The beginning of the comment is generated by the /∗ at the beginning, and the end is generated by the $*^+/$ at the end.
- The part in the middle (the Kleene closure of $([\sim*]^*|(*^+[\sim/*]))$) does not generate strings containing "∗/" because it requires that any '∗' generated inside it to be followed by something other than a '/'. The only way a '∗' can be generated is by the $*^+$ in the middle, which *must* be followed by something that is not a '/'. Furthermore, since the something that must follow any '∗' must also not be another '∗', we know that this substring cannot end with a '∗', and therefore cannot be paired with a '/' that might appear at the start of the next substring generated under the Kleene closure.
  There is an alternate formulation of the middle part that would disallow strings starting with '/', and allow strings ending with '∗', and then we would need to have a special case for any initial '/'s in the comment, and we would not need the last part above, which treats trailing '∗'s as a special case.