Week 4 Practice Exercises
Due: Noon, Monday 2/3/25

Your task is to answer the problems in this file. You should place your answers in a plain text file called `exercises4.txt` and submit it to the Canvas dropbox. Do not submit any other files. There's no need to put your name in the source code; Canvas will tell us which submission is yours. **Make sure to click the submit button**; the submission isn't registered until then. You can resubmit as many times as you like up until the deadline; we'll only look at the last submission.

You will receive credit for each problem you **attempt**, even if your solution is not correct. Some problems (or parts of problems) are marked a "optional challenge;" this means they are a bit more difficult than the others, and you are *not* required to attempt them in order to get full credit for a "good faith effort" on this submission.

Regular Expressions

To help understand how REs function, it is helpful to use an RE interpreter, which checks an input string against an RE and reports whether or not it matches. There are many such interpreters available on the web; the one used in class was `https://regex101.com`. The interface of this tool is fairly self-explanatory: you enter an RE in the top line and a test string in the box below; the match information is displayed in a panel on the right, in real time as you type.

A few things to note when using this tool:

- Regex101 supports a number of flavors of regular expressions, including Python, Java, PCRE2, etc. These are all (substantial) supersets of the version of REs described in lecture. For our exercises, you should use just the basic operators described in lecture, which are available in all modes. We suggest using Python mode, since it requires fewer escape characters than some of the others.
- By default, Regex101 tries to find all the substrings of the test string that match the RE. This is fine for many purposes, but if you want to see if the *entire* test string matches the RE, you should prefix the RE with a caret character (`^`) and suffix it with a dollar sign (`$`), and turn off the `m` option (by clicking the green characters at the right of the RE box).
- Regex101 is space-sensitive. A space within a regular expression is just like any other ordinary character: it matches a single space.

Another general point to keep in mind about regular expressions is that the basic operators have different precedences: grouping with parentheses has the highest precedence, followed by repetition, followed by sequencing, followed by alternatives. So when you examine a regular expression, it is helpful to mentally insert implicit parentheses to clarify its structure. For example, the expression `ab*c|de` is equivalent to `((a(b*)c)|(de)`.

Problem 1

Write regular expressions that will match precisely those strings that:

(i) start and end with `top`

(ii) contain either `top` or `pot` (or both)

(iii) contain both `top` and `pot` (in either order)

(iv) do not contain `top`

Problem 2

C has three forms of integer constants: decimal, octal, and hexadecimal. The constant 30 is expressed as `30`, `036`, and `0x1e` in these forms, respectively. We showed the form for decimal constants in lecture:

```
    decimal = 0 | [1-9][0-9]*
```

Give regular expressions for C octal and hexadecimal constants. (The latter was the exercise started in class.) You may want to consult the web for details about exactly what these look like.

Problem 3 (Optional Challenge)

Block comments in the C language begin with "`/*`" and end with "`*/`" and can span multiple lines. Write a regular expression that matches these block comments. (This is essentially one of the problems posed in class.) This is not so easy! Watch out for nasty corner cases such as "`/***/`". When testing your solution with Regex101, be careful of the fact that "`*`" is an RE operator; to match a literal "`*`" you typically need to escape it with a backslash, i.e. "`\*`". However, inside character class brackets, you do *not* need to escape it; e.g. "`[^*]`" matches every character *except* `*`.

Context-Free Grammars

Similarly, for context-free grammars (CFG's), it is useful to have a tool that lets us check a test string against a CFG and reports whether or not it is in the language generated by the CFG. This process is called parsing. A useful web tool for parsing is CFG Developer at `https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/`. This tool is also fairly self-explanatory to use. You first enter a grammar definition in the "Create" section at the top. The "Verify" section allows you to check that the definition is what you expected, and also displays some random test strings generated by the grammar. Specific test strings can be entered in the "Test" section; if the string matches the grammar you can click on "See Derivation" to see a (left-most) derivation—or more than one, if the grammar is ambiguous and your test string has multiple different parse trees.

A few things you need to know to use CFG Developer:

- It uses the symbol ϵ (the Greek letter epsilon) to represent an empty right-hand side for a grammar production or an empty test string. The use of ϵ for this purpose is quite common when working with formal languages, because empty strings are hard to see!
- It follows the EBNF convention of separating multiple possible right-hand sides for a given nonterminal by vertical bars (|) representing "or". You type that character in a "Create" box in order to define a new right-hand-side alternative.
- Any character can be used as a non-terminal on the left of a production. It is conventional to use capital letters, but this is not enforced. The start symbol is fixed to always be `S`. The terminals may include any character that is *not* defined as a non-terminal. (Note that this means it is impossible to use the character `S` as a terminal!)
- It is space sensitive: if you include a space on the right-hand side it is treated like an ordinary terminal, i.e. it matches a literal space.

Use CFG Developer to explore the behavior of the following CFG's. This tool is particularly useful for discovering strings that have multiple parse trees, thus proving that a grammar is ambiguous. Note that the tool cannot in general be used to prove that a grammar is unambiguous if its language is infinite: no matter how many strings with unique parse trees that you find, there might still be a string with multiple parse trees waiting to be discovered. (In fact, it turns out there is no algorithm for determining whether an arbitrary given CFG is ambiguous or not; this is an example of an uncomputable problem!)

Problem 4

Consider the following CFG:

```
E -> E + E | x
```

where `+` and `x` are terminals.

(i) Describe the language defined by this grammar in English.

(ii) Show that this grammar is ambiguous by finding a sentence that has two distinct parse trees.

(iii) Find an unambiguous grammar that define the same language.

(iv) It turns out the language defined by this grammar is regular. Find a regular expression for it.

Problem 5

Consider the set of expressions consisting of balanced parentheses: `{`ϵ`,(),(()),()(),((())()),()(()())(),...}`.

(i) Do you think this set is regular (i.e. can you define it by a regular expression)? Why or why not?

(ii) Provide a context-free grammar for this set.

(iii) (Optional challenge) Give a convincing argument that your grammar is unambiguous. (Of course, if the one you first wrote down is ambiguous, you'll need to fix it first!)

Problem 6

Using the arithmetic grammar described on Week 4 lecture slide 54 as a model, give an unambiguous grammar for boolean expressions. You should support the operators `!` (not), `&&` (and), and `||` (or), in decreasing order of precedence. The binary operators should both be left-associative. The atoms should be variables `x` and parenthesized expressions.

Analysis

Problem 7

Arithmetic operators (`+`, `-`, `*`, `/`), logical operators (`and`, `or`, `not`), and equality comparison operators (`==`, `!=`), and relational comparison (`<`, `<=`, `>`, `>=`) are four common operator categories. In most languages they may appear in a single expression, thus it is important to know their precedence.

Exercises:

(a) Give an example of a valid single Python expression that uses at least one operator from each of the four categories, and without parentheses. Evaluate the expression to a value.

(b) For both Python and C++, find the precedence order for the operators listed in the four categories above. Answer by making a clear precedence list from high to low for these operators in each language. (The actual names for some operators differ between these two languages.)

(c) Do you see any differences between the two lists? If so, provide one or two expression examples, such that each expression evaluates to a different value in the two languages.