CSC505 Jennings
Homework 4, Spring 2020
Please read, research, and answer all 6 questions below.

This assignment MUST be done alone, NOT in a group. You may NOT discuss the questions with other students. If you use any resources (e.g. websites) to answer a question, you MUST cite it there. Turn in the assignment on Gradescope.

I. Overview:

This is an INDIVIDUAL assignment. Do not discuss it at all or share information about the questions or answers with anyone.

In this assignment, you will work with Regular, Context-Free, and PEG grammars. Because text processing is so common, it is important to understand these formal models, because they describe *patterns* of text which are much more sophisticated than the substring search patterns we looked at in KMP and Boyer-Moore. (For substring search, the pattern is simply a literal string, i.e. there is no way to specify variations to match.)

For this assignment, you can turn in a PDF in almost any format – there is no template to follow. On Gradescope, you will have to indicate where in your PDF we will find the answer to each question.

Do not forget to cite any resources you used, including our textbook.

II. Questions:

1. A regular grammar[1], like a regular expression, is a concise way to denote a set of strings, where the size of the set may be infinite. Devise a regular grammar for each of the following regular expressions, using S as the starting non-terminal. The first one is done for you.

a. x*

$S \rightarrow \epsilon \mid xS$

Here is the equivalent answer without using the vertical bar notation:

$S \rightarrow \epsilon$

$S \rightarrow xS$

b. z+

c. a|b

d. 01 *(You will have to introduce another non-terminal in this problem.)*

e. (01)*

f. (01)* | 1 *(This one is tricky. Be sure your solution accepts "" and does not accept "011".)*

[1] A good reference is https://en.wikipedia.org/wiki/Regular_grammar. The regular grammar form that I showed in class is the variant called a *right regular grammar*. Also, in class I used the vertical bar symbol to indicate choices. You are free to use the vertical bar or have multiple rules for the same non-terminal (on the left hand side).

2. To demonstrate that a grammar accepts a string, we have to produce a *derivation*, which is a sequence of steps that start with the grammar's starting non-terminal (usually called S). The sequence ends with the given string. Each step is the application of one grammar rule. Recall that a grammar rule is applied by substitution. The first one is done for you.

a. Show that the grammar below accepts "xx".

S → ϵ | xS

$S \Rightarrow xS \Rightarrow xxS \Rightarrow xx$

Note that we use a double arrow to distinguish derivations from grammar definitions. Also, in the last step of this answer, S disappears because we used the rule S → ϵ.

b. Show that the grammar below accepts "abab".

S → aB | ϵ

B → bS

c. Show that the grammar below accepts "01110".

S → 0B

B → 1B | 0C

C → ϵ

3. Derivations may be constructed for any grammar, including a context-free grammar like the ones below.

a. Devise a derivation for the grammar below, showing that it accepts this palindrome string: abbcbbcbba

S → aSa | bSb | cSc | ϵ

b. Devise a derivation showing that the grammar below, which is similar to the one above, accepts the string: abca

S → XSX | ϵ
X → a | b | c

- In your derivation, make sure that you applied only one rule in each step.
- When you have an expression with multiple non-terminals, like XSX, you have a choice of which non-terminal to substitute in the next step. If we all make the same choice, then we will all have the same answer (up to grammar ambiguity). The convention we will adopt is to substitute the left-most non-terminal first. For example: XSX ⇒ aSX

4. As you know, context-free grammars are used frequently to formally define the syntax of programming languages and other text formats designed to be processed by a machine (i.e. a program will parse them). BNF[2] is a common notation for grammars. This notation uses only ASCII symbols, which made them easy to use in plain-text documents from the early days before Unicode adoption. You will find BNF in many specification documents. For example, RFC 1738[3] defines the syntax for a URL. Here is an excerpt from section 3.3, given in a variant of BNF:

```
An HTTP URL takes the form:

   http://<host>:<port>/<path>?<searchpart>
```

As you might guess, angle brackets are used here to mark non-terminals. In this example, then, "http://" is a string of literal characters, and "<host>" is a non-terminal. Whitespace can be ambiguous in this notation, however. Proper BNF, by contrast, puts quotation marks around literal characters.

In RFC1738 in Section 5 you will find a complete definition of URL, given in another variant of BNF that is closer to proper BNF. Your task is to use the RFC and the BNF definition to understand the differences.

Convert the definition of httpurl in section 5 of RFC1738 into the BNF format. You should assume that <host> and <uchar> are already defined for you to use. The first step, <httpurl>, is done for you.

- If you introduce any new non-terminals, give them upper-case names to separate them from those used in the RFC.
- Use "" instead of epsilon.
- BNF does not have parentheses for grouping.
- Do not create alternatives that have the same prefix, such as:
  `<hostport> ::= <host> ":" <port> | <host>`

Please continue the following definitions, starting with <hostport> and continuing on to define every non-terminal needed (except <host> and <uchar>) for a complete definition of <httpurl>.

```
<httpurl> ::= "http://" <hostport> <MAYBE_PATHSEARCH>
<MAYBE_PATHSEARCH> ::= "/" <hpath><MAYBE_SEARCH> | ""
<MAYBE_SEARCH> ::= "?" <search> | ""
<host> is defined separately
<uchar> is defined separately
<hostport> ::= ...
```

[2] https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form
[3] https://tools.ietf.org/html/rfc1738

5. In the Wikipedia article on BNF, the syntax of BNF is defined in BNF. Look over this definition and answer the following questions.

a. Which non-terminal is the start rule for parsing BNF?

b. The name of a rule (i.e. the name of a non-terminal) must begin with what?

c. After the first character, what else can be in a rule name?

d. How many characters can be in a rule name?

e. How many ways are there to write literals (describe them in English)?

f. How would you write one double-quotation mark, “, as a BNF literal?

g. How would you write a literal double-quotation mark, “, followed by a single quotation mark, ‘, in BNF?

6. Parsing Expression Grammars[4] are structurally similar to BNF, but they add regular expression operators like *, +, and ?. As we saw in class, these operators are *possessive* in a PEG. Also, PEGs use *ordered possessive choices* which are written using a slash, /, instead of a vertical bar as a reminder that it differs from BNF and CFG, which have un-ordered non-possessive choice operators. RPL is a PEG grammar. An example from the Wikipedia page on PEGs might be written in RPL like this, where the parentheses are a grouping construct:

```
S = ( "if" C "then" S "else" S ) / ( "if" C "then" S )
```

a. In the first BNF problem in this homework, we emphasized that you should not write BNF choices that begin with the same prefix. Here in this PEG example, we see exactly that! Both alternatives start with **`"if" C "then" S.`** Why?

b. Suppose that C were defined very simply, such that it matched any parenthesized expression, and that S also had the following rule:

```
S = "print" "(" [0-9]+ ")"
```

Give a derivation using the PEG grammar above for the following code fragment:

```
if (test1) then if (test2) then print(3) else print(0)
```

c. In the derivation you made for the previous part, there must be only one instance of S that has an "else" clause, because there is only one "else" in the code fragment. Circle the instance of S that eventually expands into an entire if-then-else statement.

[4] https://en.wikipedia.org/wiki/Parsing_expression_grammar