CS61B Lab #5

Please finish this lab today, preferably during the lab period. It is due at midnight tonight.

1. (Quiz) A Quick Review of Dynamic Method Selection

Do Quiz #0 on the lab page entry for Lab #5.

2. Scanners

First, you might want to review the online documentation for the classes java.util.Scanner, java.util.regex.Pattern, java.util.regex.Matcher, and java.util.regex.MatchResult. The code for this lab includes a file Matching.java, which you can use to play around with patterns. Compile it, and run it like this:

```
$ java Matching
Some string///
Some pattern///
Another, two-line
string///
Another pattern///
...
QUIT///
```

For each string/pattern Pair, Matching will tell whether the pattern matches the string and what each of the pattern's parenthesized groups matched.

WARNING: Inside a Java program, you use String literals to denote Strings that you want to use as patterns. String literals give the '\' character a special meaning (it is the "escape" character), so that you actually have to write "\\" in a string literal to get a single "\". When Java READS patterns, as is the case for the Matching program, it does NOT treat "\" specially, so when using Matching, you do NOT double the "\" characters (unless, that is, you want to write the pattern for a single, literal backslash character, which is "\\").

Try the second quiz (#2, there is no 1; go figure).

Last year, we had a puzzle-solving problem that involved having a program read sentences of (among others) the forms:

```
<Name> is the <Occupation>.
<Name> is not the <Occupation>.
```

where <Name> is any capitalized sequence of letters and <Occupation> is any lower-case sequence of letters. The individual words in a sentence may be separated by any non-empty amount of whitespace (blanks and tabs only, no newlines). Likewise, multiple sentences on a line may be separated by any amount of space.

Fill in the program ReadFacts.java supplied in this lab so that it will read each statement from a file that is supposed to contain nothing but blank lines and lines consisting of one or more complete statements of these forms. If you are successful, the program, when run with

```
java ReadFacts SOMEFILENAME
```

should read all the statements in file SOMEFILENAME and print back the sentences, one per line, with all whitespace standardized to a single blank. Be careful not to get into an infinite loop when you reach the end of a line. When the rest of the line does not start with a valid sentence, you'll have to figure out how to check that the rest of the line is truly blank and how to get on to the next line [hint: reading the documentation on Scanner carefully is likely to help.]

We supplied sample input and output in the files stmts.txt and stmts.out.

3. Enumerating Permutations

[This is an exercise in converting an algorithm into code.]

Project 1 last year involved solving puzzles in which N people with N different occupations live in N different houses, and one is supposed to decide who lives where and does what, given various facts (like "The carpenter does not live in the brown house.") One way to solve such puzzles is by brute force, which involves generating every possible matching of name, occupation, and house color, and testing each. One way to do that involves generating *permutations* of a list or array. For example, if we have the list (0, 1, 2), then its permutations are

```
(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)
```

Suppose we want to generate a sequence of all permutations of some range of numbers (0 to N-1 for some N). We can imagine starting with (0, 1, 2) and then repeatedly finding the "next permutation", whatever that might mean.

Here's one method, in the abstract, to get the next permutation of an an array A of length N. In the following, k and v are integer variables and S is a set of integers. The routine returns true and permutes A if it finds a next permutation, and returns false and does not change A if there is no next permutation. The "smallest" or "first" permutation of A is the sequence of integers 0 to N-1 in ascending order.

```
next_permutation(A):
   k = N-1
   S = { }
   while k >= 0:
       if S contains a value larger than A[k]:
           v = the smallest member of S that is larger than A[k]
           remove v from S
           insert A[k] in S
           A[k] = v
           A[k+1:N-1] = the values in S in ascending order.
           return true
       else:
           insert A[k] in S
           k -= 1
   return false
```

See if you can turn this abstract algorithm into reality in the file Perms.java.

APPENDIX: Summary of Scanner and Related Classes

[This is here for reference, in case it's useful. Actually, it's left over from last year. It's not part of the lab.]

A1. More on Scanners

The class 'java.util.Scanner' gives you a way to read substrings of text

("tokens") sequentially from a stream of text that is furnished to the Scanner by its constructor. Typically, the stream of text comes from a file or from a terminal, but there are ways to convert any source of characters into a stream that a 'Scanner' can process.

One constructor accepts an 'InputStream'---a stream of bytes (8-bit characters). Since 'System.in', which is normally the _standard input_ stream to your program, is a kind of 'InputStream' (that is, its type is a subtype of 'InputStream'), you can write

```
    java.util.Scanner inp = new java.util.Scanner(System.in);
```

to get something that scans the input from your terminal. (Normally, of course, you'd put

```
    import java.util.Scanner;
```

at the beginning of your source file and just write 'Scanner' instead of 'java.util.Scanner').

The simplest uses of 'Scanners' treat the input stream as a sequence of tokens separated by text that matches a _delimiter pattern_. By default, the delimiter pattern matches stretches of whitespace (blanks, tabs, newlines, carriage returns). Here are some of the common 'Scanner' methods on these token streams (assume that 'inp' is the 'Scanner' defined above):

* 'inp.hasNext()' is true iff there is another token (that is, something other than a delimiter) before the end of the input.

* 'inp.next()' returns the next token, and advances 'inp' past it.

* 'inp.hasNextInt()' is true iff 'inp.hasNext()' and the next token has the syntax of a (possibly signed) decimal numeral.

* 'inp.nextInt()' does a 'inp.next()' and then parses the token into an 'int'.

* Likewise, there are 'inp.hasNextDouble()', which returns 'double' values, and several other similarly named methods for other types.

* 'inp.hasNextInt(RADIX)' is true iff the next token exists and has the syntax of a (possibly signed) base-RADIX numeral.

* 'inp.nextInt(RADIX)' reads the next token as a base-RADIX numeral.

* 'inp.hasNextLine()' is true iff there is any more input.

* 'inp.nextLine()' returns the next line of input (that is, everything up to, but not including, the next end-of-line character, or the end of the input if there isn't an end-of-line at the end). It then positions 'inp' past the end-of-line character. Thus, it differs from the other 'next' methods in that it uses a different delimiter (end of line instead of whitespace).

Another constructor gives you a 'Scanner' that uses a 'String' as its source of characters:

```
    Scanner inp = new Scanner("Hello world!\nMy name is Jack.\n");
    while (inp.hasNext())
       System.println(inp.next ());
```

prints

```
    Hello
    world!
    My
    name
    is
    Jack.
```

A2. Patterns
----=-------

This section is just a description of Patterns and Matchers. There's nothing to do here; it's all reference

A2a. The Pattern type and pattern Strings

A 'Pattern' (full name 'java.util.regex.Pattern') describes a set of 'Strings' that it is said to _match_. Although they are sometimes called _regular expressions,_ Java 'Patterns' are actually much more powerful than formal regular expressions (which you may encounter later in upper-division CS courses).

The term "pattern" refers both to a Java object of type 'Pattern' and to a string that _denotes_ a pattern. For example, the pattern string "(jack|jill) went (up|down) the hill" denotes a pattern that matches any of "jack went up the hill", "jack went down the hill", "jill went up the hill" or "jill went down the hill". Some Java functions will take such a pattern string directly and match things with it. You can also "compile" the string into an official Java 'Pattern' like this:

```
   Pattern jackOrJill = Pattern.compile("(jack|jill) went (up|down) the hill");
```

and subsequently use 'jackOrJill' to match with. This latter choice is useful when you intend to reuse a pattern many times, since Java's internal representation of 'Pattern' is "pre-digested" and requires less execution time to interpret.

There is an annoying problem with Java Patterns. Their syntax is borrowed from that of numerous other languages with a similar feature (notably, Perl). Because they are not built into the language kernel, Java "re-tasked" Strings to serve as descriptions of Patterns. The traditional syntaxes for Pattern-like things use "\" as one of the meaningful characters. Unfortunately, it already HAS a meaning in Java String literals as an escape character. So, confusingly, wherever we say that a certain Pattern has a '\' in it, the corresponding Java String literal has "\\", since that's how you denote a (single) '\' in a String literal. So, where we would write /\d+/ in Perl or r'\d+' in Python, in Java, we write "\\d+". Believe me, it is normal to be confused at first.

The full pattern language is quite rich, and is documented under 'java.util.regex.Pattern' in the on-line Java library documentation. Here are just a few:

   * Most characters (letters, digits, most punctuation) simply match themselves.
   * A period (".") matches any character other than (usually) newline. To get "." to match newline as well, include ''(?s)'' at the beginning of your pattern. It matches the empty string, but causes '.' to match everything thereafter.
   * A sequence such as "[abe]" denotes a _character class_, in this case, "any of the (single) characters 'a', 'b', or 'e'". As a shorthand, you can represent a range of characters with a hyphen, as in "[abd-qs-z]" to mean 'a', b', 'd' through 'q', and 's' through 'z'.
   * A sequence such as "[^abe]" matches any single character other than those listed.
   * There are several useful two-character shorthands for certain character classes. '\d' is short for '[0-9]', '\s' is short for [\t\n\r] (that is, for whitespace). Unfortunately, in order to

put an actual '\' in a string, you must double it. Thus, a pattern that matches any two-digit string would be written as the string literal '"\\d\\d"'.

* If 'P' represents a pattern, then 'P*' represents "0 or more repetitions of P". Thus, 'x*' matches the empty string, "x", "xx", "xxx", etc. '[a-c]*' matches "", "a", "ab", "aa", "bac", "ccc", etc. Since "'*'" binds most strongly, so "ab*" means "one 'a' followed by 0 or more 'b's". To get the effect of "0 or more 'ab's", use '(ab)*'
* Similarly, P+ means "1 or more Ps."
* 'P?' denotes an optional P (0 or 1 Ps).
* If P and Q denote patterns, then 'P|Q' denotes "a P or a Q" (as illustrated earlier in this section).
* '(P)' denotes the same thing as P. It also serves to define a _group_, a subpattern whose match you can retrieve later.
* '(?:P)' also denotes the same thing as P, but it does not define a group that you can retrieve later.
* Following a ''*'', ''+'', or ''?'' with a ''?'' creates a "non-greedy" version, meaning that it matches as few characters as possible to make the match work. This affects what part of a string gets matched, but usually not whether a string gets matched. For example, if you are matching the string "1, 2, 3, 4" against the pattern string "(\\d).*(\\d).*", the first group will match '1' and second will match '4'. But with the pattern string "(\\d).*?(\\d).*", the second group will match '2'.
* _Boundary matchers_ match the empty string, but only at certain places. '^' and '$' match the beginning and end of a string, respectively (but see below). '\G' matches the point at which the last match ended. '\b' matches a "word boundary", a place where a word begins or ends.
* The sequence '(?m)' always matches the empty string, but has a side effect of causing '^' and '$' to match the beginnings and ends of lines as well as of entire strings.
* The two-character escape sequences '\?', '*', '\.', '\+', etc., match the character after the backslash, ignoring their special significance. Thus, the pattern 'who\?' matches the string "who?", and would be written in a program as the string literal "who\\?".

A2b. Simple pattern matching operations

The most straightforward way to use pattern matching is probably the 'String.matches' method. If aString is some string and aPattern is a string that denotes a pattern, then 'aString.matches(aPattern)' is true iff the aPattern matches all of aString. So, for example,

```
"The first thing we say is 'Hello, world'".matches (".*Hello.*")
```

is true.

A2c. Groups

The Java library gives much more than simple yes/no pattern matching, although nothing with such simple syntax. In particular, you often want to extract pieces of aString that match subpatterns of aPattern, Suppose, for example, that you want to see if your string matches the syntax of a Point "(X, Y)" and at the same time extract the X and Y values, which are supposed to be integers. Here's a method that does so, given a string printedPoint:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
...

void describePoint (String printedPoint) {
   /** A Matcher for printedPoint against the pattern
    *      \((-?\d+),\s*(-?\d+)\)
    */
   Matcher mat =
      Pattern.compile("\\((-?\\d+),\\s*(-?\\d+)\\)")
          .matcher(printedPoint);
   if (mat.matches()) {
      System.out.printf("X coordinate is %s; Y coordinate is %s.%n",
                        mat.group (1), mat.group (2));
   } else {
      System.out.printf ("Not a valid Point.");
   }
}
```

First, we compile a pattern and then create from it a 'Matcher', which incorporates the 'Pattern' and a string to apply it to. The '.matches' method matches the given string to the given pattern and saves the portions that match the two parenthesized portions. Finally, the '.group' method returns the parts of printedPoint that match the parenthesized portions of the pattern (numbering from 1 this time). 'mat.group(0)' is what matches the entire pattern. In this example, we had to indicate that the outer parentheses were real parentheses, and not grouping syntax; for that, we put a backslash in front of each (written in string literals as two backslashes).

Sometimes, a group won't be matched to anything in a particular application of '.matches()'. It is then null:

```
Matcher mat = Pattern.compile("x=(\\d+)|y=(\\d+)").matcher ("x=15");
mat.matches();
if (mat.group(1) != null)
   System.out.printf("The value of x is %s%n", mat.group(1));
if (mat.group(2) != null)
   System.out.printf("The value of y is %s%n", mat.group(2));
```

This will print "The value of x is 15" and the second printf call will not be executed.

A2d. Non-capturing groups

Sometimes, you need to group some text together in a pattern, but aren't interested in using '.group' to fetch what it matches. That's the purpose of the ''(?:...)'' syntax in patterns. Such parenthesized segments are ignored by 'group()'. For example,

```
String str = "15 greater 6";
Matcher mat = Pattern.compile("(\\d+)(?:\\s|greater)*(\\d+)").matcher(str);
if (mat.matches()) {
   System.out.printf("First number is %s and second is %s%n",
                     mat.group(1), mat.group(2));
}
```

prints "First number is 15 and second is 6".

A3. Patterns and Scanners

There are a few more methods in the 'Scanner' class that allow one to match or search for patterns in the input. Assume that 'inp' is a 'Scanner' in the following:

* 'inp.useDelimiter (DELIM)'
  causes 'inp' to use the pattern DELIM (either a string or a

```
     ‘Pattern’) as the separator between the tokens returned by
     ‘.next()’ and similar methods.  For example, after
     ‘inp.useDelimiter(",")’ tokens will be separated by single
     commas, and the input

        dog, cat,,
        pretzel sticks

     will result in the tokens ‘"dog"’, ‘" cat"’, ‘""’, and
     ‘"\npretzel sticks\n"’.

 * ‘inp.findWithinHorizon (PATN, 0)’
     reads the rest of the input (ignoring the delimiter entirely)
     looking for the first string that matches PATN (a string or
     ‘Pattern’).  Replacing 0 by some integer LIM limits the search to
     the next LIM characters of input.  This returns the matching
     string or ‘null’ if there is no match.  Also, if the pattern
     contains parenthesized groups, you may retrieve them with
     ‘inp.match().group(NUM)’.

WARNING: ‘findWithinHorizon’ is tricky to use, especially
interactively.  If the desired pattern is not in the input, it will
read the entire input, up to the end of file, trying to find it.
If you happen to be reading from the terminal when this happens, your
program will appear to hang, gobbling up all input you type in, until
you signal end-of-file to the shell or terminate your program.
```