# CS211 Spring 2007
# Assignment 2 — Lists, Recursion, Parsing
### Due Sunday, 25 February 2007, 11:59:59pm

## 1 General Instructions

### 1.1 Purpose

This assignment will help you solidify your knowledge of Java and object-oriented programming, and introduce you to induction, recursion, and list manipulation. For basic Java help, the Java Bootcamp notes on the course website, the textbook, and the online Java API are good sources.

In the first assignment, we were rather explicit in the design of the programs to help you get started. In this assignment, you will have more discretion over the design of your program.

### 1.2 Grading Criteria

As before, solutions will be graded on both correctness and style. For correctness, at the very least your program must compile without errors or warnings. For style, we like concise, clear, easy-to-read code. Use mnemonic variable names. Use proper indentation. Comment where necessary for clarification, but do not overcomment. Be concise; a two-page solution to a problem that can be done in a few lines will lose points, even if correct.

Occasionally, bonus points will be awarded for implementing optional features. Bonus points do not count in your score, but are counted separately.

### 1.3 Partners

In this assignment, you will be allowed to work in pairs. Please follow the instructions on the website for selecting a partner. You must create a group in CMS for you and your partner.

## 2 Induction

**Purpose:** Working with the induction principle
**Estimated time:** 4 hours
**Points:** 25

Prove the following statements by induction. The problems are ordered roughly in order of increasing difficulty.

(i) For all $n \geq 1$, $3^n > 2^n$.

(ii) For all $n \geq 1$, $1 + 3 + 5 + \cdots + (2n - 1) = n^2$.

(iii) For all $n \geq 1$, $1^2 + 4^2 + 7^2 + \cdots + (3n - 2)^2 = n(6n^2 - 3n - 1)/2$.

(iv) Evil mutant puddles of ooze from Oozeland have the special ability to divide themselves at will. However, there are restrictions. Each ooze can only split into 4 or 8 oozes at a time. The King of the Oozes, Bob, believes that even given this restriction, it would be possible for him to split into any number of oozes greater than or equal to 13. However Bob doesn't know math really well, and would like to prove that this is true to ooze his mind. Prove by induction that any number of oozes greater than or equal to 13 can be had.

(v) For all $n \geq 1$, $\dfrac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2 \cdot 4 \cdot 6 \cdots 2n} \leq \dfrac{1}{\sqrt{2n+1}}$.

For each one, make sure you tell us:

- the variable (usually $n$) that you are doing induction on;
- the statement you are trying to prove, preceded by the values for which it is supposed to hold, typically something like, "For all $n \geq 0, \ldots$" or "For all $n \geq 8, \ldots$" or "For all multiples of 3 greater than 17, ... ";
- the base case(s);
- the induction hypothesis;
- the induction step, indicating clearly where you use the induction hypothesis;
- the conclusion.

Structure your proofs carefully and identify all these parts clearly.

Submit your answers in a plain text file `Induction.txt`. Do not use a text formatter and do not format your text. For the exponentiation operator, use `^`. For the multiplication operator, use `*`. Use `pi` for $\pi$. For the square root, use `sqrt( )`. Parenthesize expressions to avoid ambiguity; for example, $2^{n+1}$ should be written `2^(n+1)`.

# 3 A File System Simulation

**Purpose:** List manipulation, simulation
**Estimated time:** 10 hours
**Points:** 30

For this part of the assignment, you will be modeling a simple file system. A files system includes two basic things: a collection of files and a directory structure. All file systems have a few basic operations, such as creating a file, accessing a specific file, and deleting a file from the collection.

## 3.1 Implementing the File System

Our file system will consist of a linked list. Each node in the list will represent a single file. The data contained in each node of the list should be a `String` representing the file name. (For this problem, you do not have to represent the contents of the file.) Unlike Windows, filenames in our system are case-sensitive; that is, `HELLO`, `Hello`, and `hello` are to be considered three different files.

Your file system will be implemented in a public class `FileSystem`. This class must contain six public methods with the signatures exactly as shown and the indicated behavior:

- `public void create(String name) throws IllegalArgumentException`
  If there does not already exist a file in the system with the given `name`, create a new file with that `name` and insert it in the linked list at the head of the list. If the file already exists, throw an `IllegalArgumentException`, but do not print any message.
- `public int access(String name) throws NoSuchElementException`
  Find the file with the given `name` in the file system. You will need to search the list from the front. If the file does not exist, throw a `NoSuchElementException`, but do not print any message. If the file was found, return an `int` giving the position in the list where you found it, starting at 0. This number is the same as the number of items in the list in front of the accessed element.
- `public int delete(String name) throws NoSuchElementException`
  Exactly the same as `access`, except delete the file from the file system after you have found it.

- `public int accessBubble(String name) throws NoSuchElementException`
  Exactly the same as `access`, except that after you have found the file, swap it with the file immediately in front of it on the list. Do this by unlinking the two elements and swapping their positions, not by just reassigning the names in the list elements. For example, if the file system contains `A => B => C => D`, and `accessBubble("D")` is called, then it should return 3, and afterwards the system should look like `A => B => D => C`. If the accessed element is already at the head of the list, don't do anything.

- `public int accessMoveToFront(String name) throws NoSuchElementException`
  Exactly the same as `access`, except that after we have found the file, we will move it to the front of the list. For example, if the file system contains `A => B => C => D`, and `accessMoveToFront("D")` is called, then it should return 3, and afterwards the system should look like `D => A => B => C`.

- `public String toString()`
  Return a string representation of the entire file system with the file names in order separated by `=>`, as shown above.

The list modifications in `accessBubble` and `accessMoveToFront` are justified as follows. In the worst case, searching through a linked list takes $O(n)$ time, also known as linear time. We will discuss big-$O$ notation later in the course, but the idea is that every time we search through the list, it could take as many as $n$ steps to find the desired file, where $n$ is the number of files in our system. However, in real systems, file accesses tend to exhibit *locality of reference*, which means that they tend to be clustered. There is a good chance that a file that has been accessed recently will be accessed again very soon. For example, if we have just read a buffer of data from a file, perhaps a short time later we will want to read another buffer from the same file.

We can try to gain some performance advantage from locality of reference by rearranging the file system after each access. Whenever a file is accessed, we can move it closer to the front, so that if it is referenced again soon in the near future, we will not have to spend as much time looking for it.

The three access methods represent three different strategies. The `accessBubble` method represents a less drastic restructuring strategy, the `accessMoveToFront` method represents a more drastic strategy, and the `access` method represents no strategy at all.

The `int` returned by these methods should be the number of elements in front of the accessed element *before* any modifications to the list were performed. That is, it should be the number of files you had to search through before finding the desired file.

This exercise is meant to give you experience with low-level operations on lists. Thus, for this exercise, you should not use `ArrayList`, `Vector`, or any similar preprogrammed class in the Java API, but should write your own. The list should be singly-linked, not doubly-linked. Since it is expensive to traverse a list, you should make sure that any of your methods that need to traverse the list do so at most once.

Submit your code in a file `FileSystem.java`.

## 3.2 Testing Locality Strategies

Now we would like you to do some experiments to compare the three restructuring strategies on random sequences of file operations. We have provided some code that generates an endless random sequence of file operations according to a certain distribution. The operations are all of the form (*command, filename*) where *command* is one of `CREATE`, `DELETE`, or `ACCESS`. The code to generate the sequence is in the file `a2.jar` and there is full Javadoc documentation (just like in the Java API) available from a link on the website under **Assignments**. To use this code, put the `a2.jar` file someplace in your classpath and put `import a2.*;` at the beginning of your `.java` files. You can then generate a random sequence of `VERY_LARGE_NUMBER` file operations as follows:

```
FileSystemTest fst = new FileSystemTest();
for (int i = 0; i < VERY_LARGE_NUMBER; i++) {
   Operation op = fst.nextOperation();
   ...do something with op...
}
```

(you have to decide on `VERY_LARGE_NUMBER`). Write a program that tries out each of the strategies on several very long sequences of file operations produced by our code, and compute the average search time under each strategy. Report on which strategy is the best.

You can work with very long sequences (hundreds of thousands of file operations). The algorithm that generates the operations will not try to access or delete a file that has not been created or a file that has already been deleted. It is tuned to hover around an equilibrium file system size of about 360–370 files, even over hundreds of thousands of file operations.

Submit your testing code in a file `Experiments.java` and your conclusions in a text file `Conclusions.txt`.

# 4 MadLibs

**Purpose:** Recursion, parsing
**Estimated time:** 15 hours
**Points:** 40

Your childhood friend Check Steadyhammer is an aspiring writer. Unfortunately, while he has quite a way with story lines, he has a lot of difficulty thinking up imagery sufficiently creative to impress his audience (sophisticated literary critics). He thinks that may be you could help him, by having a computer program randomly fill in some parts of his stories for him, hopefully with something appropriate or at least inspiring. Now, normally, this would be a daunting task, as the English grammar is very complicated, but luckily Check is an adherent of the avant-garde Verbosist-Structurist style. This style has two important characteristics: first, there is little variation of sentence structure, making the grammar pretty simple. Second, the style insists on describing details, so many parts of speech such as adjectives and adverbs are required in most cases. This is handy as it means your program can figure out what part of speech to fill in automatically in most cases, which is especially important since you know that the grammar class was a painful experience for Check, and you would rather not remind him of it.

Your goal in this assignment is to complete a program for this scenario. We have provided much of the needed auxilary code, but it is lacking the code to parse the very restricted subset of English. In particular, you'll have to complete the `parseSentence` method of the `Parser` class, using the recursive descent technique you learned about in class. Your parser should use the following grammar:

| | | |
|---|---|---|
| <*Sentence*> | → | <*DeclarativeSentence*> terminator |
| <*Sentence*> | → | <*QuoteSentence*> terminator |
| <*DeclarativeSentence*> | → | <*NP*> <*VP*> |
| <*QuoteSentence*> | → | quote <*DeclarativeSentence*> quote verb <*NP*> |
| <*NP*> | → | determiner adjective noun <*MaybePP*> |
| <*MaybePP*> | → | preposition <*NP*> |
| <*MaybePP*> | → | $\epsilon$ |
| <*VP*> | → | adverb verb <*MaybeNP*> |
| <*MaybeNP*> | → | <*NP*> |
| <*MaybeNP*> | → | $\epsilon$ |

and return as its output the text of the completed sentence. Here $\epsilon$ represents the empty string, the symbols < . . . > are the nonterminals of the grammar, and the symbols in Roman typeface are the types of the tokens.

We have provided a tokenizer that produces a stream of tokens. Your recursive-descent parser should be able to determine the appropriate recursive call to make depending on the next token in the input stream. This can be obtained by calling `tokenizer.lookahead()`.

Besides the token types described in the grammar, there is one other special type: *blank*, which represents a word that your program must fill in. To handle the blanks, if the parser knows for sure what part of speech to expect, it should generate a random word and output it in place of the blank. If the parser cannot determine the part of speech, it should issue an error, except you should assume that a blank at the beginning of a sentence will never be a quote.

The determination of the part of speech and generation of random words are done by the `Dictionary` class, which we provided. This class in turn reads in the known words from the file `dictionary.txt`. Your

parser should read the input using the provided `Tokenizer` class, which takes care of the I/O, splitting out the words, and classifying the words using the `Dictionary` for you; but you'll have to issue descriptive parser errors on any errors the tokenizer passes to you, such as unknown words, I/O errors or unexpected end of file. The parser must also issue an error if the sentence is malformed.

We provide two test programs for your convenience. The `StoryCompleter` takes a filename as an argument and makes your parser parse all the sentences in it, outputting the completed version. You can use the files `story1.txt` and `story2.txt` as starting points for your testing, but you should try other inputs as well. The `CompleterTester` program, on other hand, is interactive, and completes one sentence at a time. Note that in either case, you can represent a blank as a sequence of underscore (_) characters in the input.

Supplied code for this problem can be found in `A2P4.zip`. You can run `Madlibs.jar`, which runs our version of `StoryCompleter`, to see how it is supposed to work. Submit your completed `Parser.java`. Also, please turn in your favorite produced output in a text file `FavoriteOutput.txt`.

# 5 Submitting Your Work

**Points:** 5

Submit the following files in CMS:

- Exercise 2: `Induction.txt`
- Exercise 3: `FileSystem.java`, `Experiments.java`, `Conclusions.txt`
- Exercise 4: `Parser.java`, `FavoriteOutput.txt`

Do not submit any other files. In particular, do not submit any `.class` files. Refer to the submission format requirements on the course website before submitting any work.