# CS2110 Assignment 2 — Lists, Induction, Recursion and Parsing, Summer 2008
**Due Thursday July 3, 2008, 6:00PM**

# 0 General Instructions

## 0.1 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.

## 0.2 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.

## 0.3 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.

# 1 Induction

**Purpose:** Working with the induction principle
**Points: 30**

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 $n \geq 1$, $\frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2 \cdot 4 \cdot 6 \cdots 2n} \leq \frac{1}{\sqrt{2n+1}}$

For each of the above problems, 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 $n \geq 0, \ldots$";
- 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}`.

# 2 Subway Survival

**Purpose:** List manipulation
**Points: 35**

In this part of the assignment, you will model the seating arrangments for passengers on a train. For simplicity, we will assume that the train has a long linear row of seating space. Let a 'sub' be the unit used to measure seating space. All through the day, several passengers board the train and alight from it. Every passenger has a width, expressed as an integral number of 'subs'. The train has a single door near the engine, and every passenger uses this door to board the train. After getting onto the train, each passenger walks the length of the train searching for space to sit. Passengers typically prefer to maintain their distance from other passengers, and consequently each passenger searches for the *largest* section of vacant space before sitting down.

The train thus contains several passengers (of varying widths) seated in a row, and consecutive passengers are usually separated by some vacant space. Occasionally, it might happen that a passenger is unable to find a section of vacant space large enough to seat him/her, and such a passenger has no choice but to put on a grim smile and stand throughout the journey. Every once in a while, it so happens that an intimidating passenger boards the train, who barks out "Move!". Upon which all the passengers shift over to the head of the train, covering up the vacant spaces between them and leaving a large vacant space at the rear end of the train.

## 2.1 Implementing the Seating system

We will represent the occupied seats and vacant spaces using a singly-linked list, and implement certain operations on this list as specified below. Each node in the list will represent a section of the seating space; either occupied by a person with a specified name and width, or an unoccupied section of vacant space. Nodes in this list would be instances of the class `Seat`, as defined in the provided file `Seat.java`.

Each `Seat` object has four fields: `personName` which is `null` if the seat is empty, else records the name of the person occupying the seat; `start` denotes the index of the starting location of the seat; `width` denotes the width of the seat; and `next` is a pointer to the next `Seat` object in the list. Note that each `Seat` object thus represents seating space indexed from `start` to `start + width - 1`. The seating space represented by the list must be continuous, i.e. the `start` value of a node must equal `start+width` as computed on the previous node.

You are required to implement the class `Subway`, that keeps track of the current state of the seating space on the train. This class must contain six public methods with signatures exactly as shown below, and must exhibit the indicated behavior:

- `public Subway(int capacity)`

  Initialize the `Subway` using the given `capacity`.

- `public int allocateSpace(String personName, int personWidth)`

  Scan the list and find the widest section of vacant space. If there is no section wide enough to accommodate the person, return $-1$. Else, create a new `Seat` node $s$ with width `personWidth` and name `personName`, and place it centred upon the midpoint of the vacant space(explained in detail below). Create new nodes to represent the smaller sections of vacant space(if any) before and after $s$, and appropriately instantiate their field values.

- `public int findPerson(String personName)`

  Scan the list and find the `start` location of the seat where `personName` is seated. Return $-1$ if `personName` cannot be found in the list.

- `public int freeSpace(String personName)`

  Scan the list and find the `start` location of the seat where `personName` is seated. Return $-1$ if `personName` cannot be found in the list. Free the space occupied by `personName` by emptying the corresponding `Seat` node $s$. Also check if the nodes adjacent to the node $s$ are empty, in which case they should be merged with $s$.

- `public String toString()`

  Generate a string representation of the contents of the list, primarily mention the `start` position and name of the occupying person(if any) for all the `Seat` nodes.

- `public void compactSeats()`

  This corresponds to the scenario where all passengers are compelled to shift in the direction of the head of the train(thereby eliminating the vacant spaces between the passengers). Appropriately modify the `start` values for all the occupied nodes, and create a single empty node at the tail of the list (that symbolizes all the vacant space on the train).

For the sake of consistency, we make the following assumptions. A train with capacity $c$ consists of a linear row of subs indexed from 0 to $c-1$, where 0 denotes space near the head of the train. In `allocateSpace()`, a passenger is seated centred exactly on the midpoint of the vacant space. Again for consistency, we assume that all passengers have odd-numbered widths, and that the middle of a section of width $w$ is determined using integer division $w/2$. For example, given a section of space indexed from $[10-21]$, and a passenger of width 5; we see that the space has size 12, its midpoint is 16, and the passenger is seated in $[14-18]$. The `int` returned by the above methods is the starting position of the allocated space(, or the found or deleted passenger.) Also note that upon merging nodes, all unused/old nodes must be de-linked from the list.

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 a list do so at most once.

Submit your code in a file `Subway.java`. You may add methods to `Seat.java` if you like, although we suspect that wouldn't be necessary. Be sure to test the methods in the `Subway` class (including the one that compacts the seats) using sequences of passengers (of different widths) that board and alight from the train.

# 3 Parsing

**Purpose:** To become familiar with recursion and parsing
**Points:** 35

## 3.1 Syntax

In this problem you have to write a parser for the language of a simple calculator which supports the four basic operations $+, -, \times$ and $\div$ using floating point numbers. The calculator also has 26 memory locations (registers) named a,b,c,...,z which can store previously computed results. The language supported by the calculated may be described using the following grammar:

```
Line → Ident '=' Exp
Exp → Term ExpRest
ExpRest → '+' Term ExpRest | '−' Term ExpRest | ϵ
Term → Factor TermRest
TermRest → '*' Factor TermRest | '/' Factor TermRest | ϵ
Factor → '(' Exp ')' | Ident | '−' Factor | float
Ident → 'a' | 'b' | ...| 'z'
```

The symbols that appear in single quotes are terminal symbols and the symbols that begin with a capital letter denote non terminal symbols. The special symbol $\epsilon$ denotes the empty string and the special symbol `float` denotes any string that can be successfully converted into a float using `Float.parseFloat`. The notation $A \rightarrow B|C$ means that $A$ can produce either $B$ or $C$. Note that the spaces between each symbol above are important. For example, `( 19 * -20 )` is different from `(19 * -20)`. Only the former is valid syntax!

## 3.2 Semantics

Every input line you give to the calculator should look similar to this example:

```
x = ( y + z ) * ( 0.5 + w )
```

The first token should be a register name, followed by `=` and then an arithmetic expression. What should happen when this input is given is first to make sure that the input is a correct expression and then evaluate the expression on the right of the = sign and assign the result in the memory location indicated by the register name on the left of the = sign. Evaluating the expression follows the standard arithmetic rules (first parentheses, then multiplications and divisions and then additions and subtractions and proceeding from left to right[1]). Following the Java convention, all memory locations initially contain 0.

Based on the syntax and semantics given above, you will need to write a parser that accepts this language, evaluates the expressions it is given and stores the results in the appropriate memory locations. This is how your program should work. First, it should be started from the command-line:

```
> java Calculator
```

At this point, the calculator will wait for your input. After you enter a valid input line the calculator should print out the non-zero memory locations. For example, you might have the following session:

```
a = 2
a: 2.0
b = a + 1
a: 2.0 b: 3.0
c = a * - b
a: 2.0 b: 3.0 c: -6.0
d = 10 - c - a
a: 2.0 b: 3.0 c: -6.0 d: 14.0
e = d / a
a: 2.0 b: 3.0 c: -6.0 d: 14.0 e: 7.0
e = e * b * a
a: 2.0 b: 3.0 c: -6.0 d: 14.0 e: 42.0
x = ( e - 12 ) / ( 1 + ( d + c ) / 2 )
a: 2.0 b: 3.0 c: -6.0 d: 14.0 e: 42.0 x: 6.0
y = 1.5 / ( x + c )
ParserError: at line 8: Division by zero
y = 1 + 1i
ParserError: at line 9:  Unxpected token 1i
y = a +
ParserError: at line 10: Unexpected end of line
```

[1] E.g. $5 - 2 - 1$ should be parsed as $(5 - 2) - 1$ not as $(5 - (2 - 1))$

```
y = ( a + b
ParserError: at line 11: Unexpected end of line
42
ParserError: at line 12: Expected register name but got 42
a <- 1
ParserError: at line 13: Expected = but got <-
a = a & b
ParserError: at line 14: Unexpected token &
a = bb + 1
ParserError: at line 15: Unxpected token bb
```

In the last line we just gave an empty line to quit.

If your calculator encounters any line of input that's not a valid expression it should output an error. Your error messages should be rather descriptive as above. You may want to make use of the try and catch statements for exception handling, though it's not required. However, your program should not simply crash/exit on invalid input. To get you started we already give you a Scanner-like class with the ability to take a sneak peek at the next token without consuming it. This will come in handy when you will be implementing the Parser. You will need to write the main method, the parser and probably some helper class that will keep track of what is stored in the registers of the calculator. Submit your solution in `Calculator.java`.

# 4 What to submit

Submit the following files:

1. `Induction.txt`
2. `Seat.java, Subway.java`
3. `Calculator.java`

# Extra Credit, Extra Fun!

If you have finished everything above and are craving for some exercise for the gray cells, here are some extra credit items you can try your hand at. *We will award extra credit based on how well they are done. However, if the answers to the main part of the homework are not well done, we will not award very many extra credit points at all.*

## 4.1 Induction Proofs

Here are a couple of more results for you to prove using induction.

(i.) $4^n + 15n - 1$ is divisible by 9 for all $n \geq 1$.

(ii.) Consider a grammar with the start symbol $A$, variables $\{A, B\}$ and rules as follows:

$A \rightarrow B$
$B \rightarrow AB$

This grammar can be used to generate the following sequence of strings:

| | |
|---|---|
| $n = 0:$ | $A$ |
| $n = 1:$ | $B$ |
| $n = 2:$ | $AB$ |
| $n = 3:$ | $BAB$ |
| $n = 4:$ | $ABBAB$ |
| $n = 5:$ | $BABABBAB$ |

and so on.

Note that to generate these strings, it is necessary that at each step, *both* rules are applied. That is, all $A$s are transformed to $B$s, and all $B$s are transformed to $AB$s. You may observe that if you count the length of each string, we obtain the sequence of Fibonacci numbers $1, 1, 2, 3, 5, 8, \ldots$.

Prove formally, using induction, that the length of the $n^{th}$ string in this sequence is equal to the $n^{th}$ Fibonacci number.

(Hint: You may use the observation that $T(s_1 \cdot s_2) = T(s_1) \cdot T(s_2)$; where $s_1 \cdot s_2$ denotes the concatenation of the two strings $s_1$ and $s_2$, and $T$ represents the transformation encoded in the two rules of the grammar.)

Start Early, and Good Luck!