

## Section Handout

---

### Discussion Problem 1: <http://tinyurl.com>

**tinyurl.com** is a website that constructs small, more easily managed URLs on behalf of long, unmanageable ones. As an example, here's a link leading to a page that outlines how to drive from Jerry's house in San Francisco to Stanford campus:

```
http://maps.google.com/maps?f=d&source=s_d&saddr=Noe+St,+San  
+Francisco,+CA+94114&daddr=Stanford+University,+Stanford,+CA  
+94305&hl=en&geocode=&mra=1s&sll=37.596475,-  
122.301758&ssp=0.35472,0.831528&ie=UTF8&ll=37.597912,-  
122.305984&spn=0.354712,0.831528&z=11
```

While Google Maps is undoubtedly a terrific product, that URL is a little much, particularly if you have to drop it into an email invitation and expect all the recipients to cut and paste that to a browser. It also makes it difficult to drop a link like that into your Facebook or Twitter status, particularly when Twitter limits you to 140 characters.

TinyURL allows you to enter a long, nasty URL like this, and maps it to a much simpler URL that can be shared with others more easily. TinyURL now maps the above to this:

```
http://tinyurl.com/dzsoq4
```

This tiny URL is essentially the same as the larger one. TinyURL manages the correlation for you.

See the trailing **dzsoq4**? You may not immediately see it this way, but that's actually a number—a base-36 number. TinyURL uses that number behind the scenes to find the original URL so it can fetch the right page for you.

Write two functions: **TinyURLConvertNumber** and **TinyURLRecoverNumber**. The first one accepts a positive integer and generates the base-36 equivalent, and the second does the opposite. In base 36 land, 0 – 9 correspond to 0 – 9, and a – z correspond to 10 – 35: these are the digits of a base 36 counting system. Woo.

```
/**  
 * Accepts the incoming number, assumed to  
 * be positive, and generated the equivalent  
 * base-36 number as a string of digits and  
 * lowercase letters.  
 *  
 * Examples:  
 * TinyURLConvertNumber(2) returns "2"
```

```

*   TinyURLConvertNumber(49) returns "1d"
*   TinyURLConvertNumber(18293001) returns "aw2yx"
*   TinyURLConvertNumber(738291002) returns "c7k4ze"
*/
string TinyURLConvertNumber(int num);

/**
 * TinyURLRecoverNumber is the functional inverse
 * of TinyURLConvertNumber. It accepts a properly
 * formatted string and returns the base-10 equivalent.
 *
 * Examples:
 *   TinyURLRecoverNumber("2") returns 2
 *   TinyURLRecoverNumber("1d") returns 49
 *   TinyURLRecoverNumber("aw2yx") returns 18293001
 *   TinyURLRecoverNumber("c7k4ze") returns 738291002
 */
int TinyURLRecoverNumber(string code);

```

### Discussion Problem 2: Maximizing Game Score

Consider the one-player puzzle where you're presented with a 2 by n board of small positive integers like the following:

5	7	1	8	4	8	3	6	2	1	1	5	1
7	1	3	3	4	9	6	1	5	9	2	3	2

You're given an unlimited number of pebbles, and your task is to distribute pebbles across the board—with at most one pebble per square—subject to the constraint that you can't place pebbles in vertically, horizontally, or diagonally adjacent locations. Your score is the sum of all of the values inside the squares covered by pebbles.

Your job is to accept a reference to the game board (expressed as a **Grid<int>**) and to return the maximum score one can get by playing that board.

```

/**
 * Returns the maximum score one can get by optimally
 * distributing pebbles across the specified game board,
 * subject to the constraints described above. The function
 * doesn't describe the optimal distribution of pebbles, just
 * the best possible score.
 *
 * You may assume that the Grid<int> has exactly two rows
 * and at least one column.
 */
int ComputeMaxScore(Grid<int>& board);

```

### Lab Problem 1: Look And Say

The look-and-say sequence is the sequence of numbers beginning as:

```
1
11
21
1211
111221
312211
13112221
```

Each number in the sequence is generated by "reciting" its predecessor, and then capturing what was said in a number format that should be clear from example. Reciting 111221 out loud, you'd look and say: 3 ones, followed by 2 twos, followed by 1 one, or 312211. Reading 312211 aloud, you'd say: 1 three, 1 one, 2 twos, 2 ones, or 13112221.

The lab code has been set up so that a main function is completely empty. You should implement the **main** function to keep printing numbers until you quit the program.

### Lab Problem 2: Scheme Expressions

Arithmetic expressions in the Scheme programming language look a little different than they otherwise would in C++. For instance, the C++ expressions:

- $1 + 2 + 3 + 4 + 5$
- $2 * 5 * 9 * 14 * 20$
- $8 - 2 - 18 - 6$
- $64 / 8 / 2 / 2$
- $(4 + 6) * (7 / 4) + 3$

look like this in Scheme:

- $(+ 1 2 3 4 5)$
- $(* 2 5 9 14 20)$
- $(- 8 2 18 6)$
- $(/ 64 8 2 2)$
- $(+ (* (+ 4 6) (/ 7 4)) 3)$

Using the **Scanner**, **Vector**, and **Stack** classes, you should write an **Evaluate** function that accepts an arithmetic Scheme expression, as a string, and returns the number it evaluates to. There is, in theory, no limit to the number or depth of parenthetical sub-expressions. If you ever encounter a parsing error, an unexpected operator, additional tokens, or a division by zero, you should just call the **Error** function and not worry about finishing the computation.

The lab code has been set up to interactively prompt the user to enter an expression and then print out its evaluation. Your job is to implement **Evaluate**.

Note that this is an optional exercise for those who breeze through everything else.