CS 135 Fall 2008
Byron Weber Becker, Ian Goldberg, Brad Lushman, Daniel Roche, Troy Vasiga

Assignment 9

Due Wednesday, November 19, 10:30am

Files to submit: `sublists.ss`, `matrix.ss`, `gcd.ss`, `translate.ss`, and for the bonus to 3(c), `bettergcd.ss`.

Language level: Intermediate Student with Lambda
Warmup Exercises: (Not to be submitted) 25.2.5, 26.1.2, *Without using recursion*: 9.5.3, 9.5.4
Extra practice exercises: (Not to be submitted) 25.2.3, 25.2.6, All of 27.5, *Without using recursion*: 9.5.5, 9.5.7

Note: Appropriate uses of the function *append* are permitted on this assignment.
Note: Except where prohibited by the question, look for opportunities to use abstract list functions to simplify your implementations.

1. Place your solution in the file `sublists.ss`. Given a list L, we say that a list L' is a *sublist* of L if L' can be obtained by removing zero or more elements from L. The remaining elements are in the same order in L' as they are in L. For example, '(1 3) is a sublist of '(1 2 3 4), but '(3 2) is *not* a sublist of '(1 2 3 4). It follows from the definition of a sublist that the empty list is a sublist of every list (including itself), and every list is a sublist of itself.

   For this question, you may assume that the list *L* mentioned below does not contain duplicate entries.

   (a) Write the recursive function *sublists-rec* that consumes a list *L* and produces the list of all possible sublists of *L*. For example, (*sublists-rec* '(1 2 3)) should produce the list

   ```
   '(() (1) (2) (3) (1 2) (2 3) (1 3) (1 2 3))
   ```

   The order in which the sublists occur is not important. Do *not* use abstract list functions to implement *sublists-rec*.

   (b) Use abstract list functions to write the non-recursive function *sublists-abs*, that behaves in the same way as *sublists-rec*. You must not use any recursion to implement this function, except as provided by abstract list functions.

2. Place your solutions to the following questions in the file `matrix.ss`. Use abstract list functions as appropriate to simplify your code. A *matrix* (*mat*) is an $m \times n$ array of numbers ($m, n \geq 1$), implemented as a list of m lists, each of length n (a matrix is called a *square matrix* if $m = n$). For example, the following is an example of a 2×3 matrix:

   ```
   (define mtrx '((1 2 3)
                  (4 5 6)))
   ```

   (a) The *transpose* of a matrix is the matrix obtained by changing its rows into columns and vice versa. For example, the transpose of *mtrx* above is

   ```
   '((1 4)
     (2 5)
     (3 6))
   ```

Write the function *transpose* that consumes a matrix and produces its transpose.

(b) **5% Bonus** You may have heard about a wonderful mathematical object known as the *determinant* of a matrix. Determinants have many special useful properties; for example, a determinant of 0 means that the matrix has no inverse. The simplest way to compute the determinant of a matrix is to perform what is called "minor expansion": Given a square $n \times n$ matrix A, sum, for $i = 1, 2, \ldots, n$, $(-1)^{i-1}$ times the product of the i^{th} entry in the first column of A times the determinant of the matrix formed by removing the 1^{st} column and i^{th} row from A.

So for example, given the matrix

$$A = \begin{bmatrix} 3 & 1 & 9 \\ 4 & 8 & 5 \\ 2 & 7 & 6 \end{bmatrix},$$

we can compute the determinant of A as:

$$\det A = 3 \cdot \det \begin{bmatrix} 8 & 5 \\ 7 & 6 \end{bmatrix} - 4 \cdot \det \begin{bmatrix} 1 & 9 \\ 7 & 6 \end{bmatrix} + 2 \cdot \det \begin{bmatrix} 1 & 9 \\ 8 & 5 \end{bmatrix}.$$

Minor expansion, along with the fact that the determinant of a 1×1 matrix is just the single entry in the matrix, gives you the ability to compute the determinant of any matrix.

Write a scheme function *det* which consumes a square matrix and computes its determinant.

3. Place your solutions parts (a) and (b) of the following question in the file `gcd.ss`. If you do the bonus in part (c), put your solution for that part into a separate file called `bettergcd.ss`.

(a) Write a function *gcd-list1* which consumes a non-empty list of integers and produces the greatest common divisor of all numbers in the list. Use abstract list functions so that your function does not use explicit recursion. You should use the built-in function *gcd*, but don't call it it with more than two arguments.

(b) The cost of computing greatest common divisors is proportional to the size of the two numbers involved. But (assuming you start with a list of numbers with the same size), your *gcd-list1* function can be very inefficient because most of the times you call *gcd* one of the arguments will be much larger than the other.

Solve this problem by writing a function *gcd-list2* which consumes a non-empty list of integers and produces the greatest common divisor by repeatedly computing gcds of adjacent elements in the list. For example, given the list '(48 84 60 90), you should first compute (*gcd* 48 84) and (*gcd* 60 90) to get 12 and 30, and then (*gcd* 12 30) to get 6 as the final answer. (Hint: you will need a helper function like *merge-all-neighbors* from warmup exercise 26.1.2.)

(c) **5% Bonus** For this bonus, you will write two improvements to your function from the previous part. Call this function *gcd-list3*. Make sure to put your solution to this part only in a separate file called `bettergcd.ss`.

First, it is easy to see that $\gcd(n, 1) = 1$ for any integer n. So if 1 is returned as a gcd at any point in the computation, we can immediately return the value of 1 without doing any more work. Your new implementation in *gcd-list3* should return 1 immediately whenever a call to *gcd* returns 1, without performing any extra computation.

The second improvement is based on the fact that $\gcd(n, m) \leq \min\{m, n\}$, that is, the size of the result is bounded by the smaller input number. Modify your implementation so that, at every recursive level, we repeatedly call *gcd* on the largest and the smallest numbers remaining in the list. So for the example above, we would compute (*gcd* 90 48) and (*gcd* 84 60) on the first level, and then (*gcd* 12 6) on the second level.

4. Place your solution to the following question in the file `translate.ss`. Residents of the planet Lambda speak a peculiar language, which is based on English, except that all of the consonants occur before all of the vowels, and consecutive identical letters are not allowed. To translate from English to the language of Lambda, you follow these three rules:

   (a) Whenever a vowel directly precedes a consonant, swap them (e.g., "cat" becomes "cta").

   (b) Whenever a vowel appears in consecutive positions, remove both of them (e.g. "cheese" becomes "chse").

   (c) Whenever a consonant appears in consecutive positions, remove the first one (e.g. "mmm" becomes "mm", and then "m" on a second application of the rule).

   It may be necessary to apply the rules more than once. For example, "mutter" above becomes "mtuter", then "mttuer", then "mtuer", then "mture", and finally, "mtrue". The rules may be applied in any order, but you must continue to apply them until all of them are no longer applicable. Write the Scheme function *translate* that consumes an English string and produces its Lambda-language equivalent. Then, in comments, give a brief, informal argument that your function always terminates. You may assume that the English string contains only lowercase letters and spaces. A space is neither a vowel nor a consonant. For the purposes of this assignment, vowels are a, e, i, o, u. All other letters are consonants.

The remaining material in this document consists of enhancements which do not need to be submitted. We start with some more mathematical material and finish with an artificial intelligence application that uses list processing (which would make a fine set of final exam practice exercises for all CS 135 students).

Let f_n, $n = 0, 1, ...$, be the Fibonacci sequence (i.e., $f_0 = 1$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$ for $n > 1$). Show that if $u = f_n$ and $v = f_{n+1}$, then (*euclid-gcd v u*) has depth of recursion n. Conversely, show that if (*euclid-gcd v u*) has depth of recursion at least n, and $u < v$, then $u \geq f_n$ and $v \geq f_{n+1}$. This shows that the Euclidean GCD algorithm has depth of recursion proportional to the logarithm of its larger input, since f_n is proportional to ϕ^n, where ϕ (the "golden ratio") is about 1.618.

You can now write functions which implement the RSA encryption method (since Scheme supports unbounded integers). You have already seen fast modular exponentiation (computing $a^n \bmod m$) in lecture module 05. For primality testing, you can implement the little Fermat test, which rejects numbers for which $a^{n-1} \equiv 1 \pmod{n}$, but it lets through some primes. If you want to be sure, you can implement the Solovay-Strassen test as follows: If $n - 1 = 2^d m$, where m is odd, then compute

$a^m \pmod n$, $a^{2m} \pmod n, \ldots, a^{n-1} \pmod n$. If this sequence does not contain 1, or if the number which precedes the first 1 in this sequence is not -1, then n is not prime. If n is not prime, this test is guaranteed to work for at least half the numbers $a \in \{1, \ldots, n-1\}$. (For more information, see `http://en.wikipedia.org/wiki/Solovay-Strassen_primality_test`.)

Of course, both these tests are probabilistic; you need to choose random a. If you want to run them for a large modulus n, you will have to generate large random integers, and the built-in function *random* only takes arguments up to 2147483647. So there is a bit more work to be done here.

For a real challenge, use Google to find out about the Miller-Rabin test (which is probably the most commonly-used primality test), and the more recent deterministic polynomial-time algorithm for primality testing due to Agarwal, Kayal, and Saxena, and implement those.

Continuing with the Math 135 theme, you can implement the extended Euclidean algorithm: that is, compute integers a, b such that $am + bn = gcd(m, n)$, and the algorithm implicit in the proof of the Chinese Remainder Theorem: that is, given a list $(a_1, \ldots a_n)$ of residues and a list $(m_1, \ldots m_n)$ of relatively coprime moduli ($gcd(m_i, m_j) = 1$ for $1 \leq i < j \leq n$), find the unique natural number $x < m_1 \cdot m_2 \cdots m_n$ (if it exists) such that $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, n$.

It is perhaps stretching to call what we are about to look at "an artificial intelligence application". It is a program which mimics a human psychologist in an extremely simple way. The idea comes from Joseph Weizenbaum, who wrote a similar program and reported on it in the January 1966 issue of Communications of the ACM. A typical interaction might look like this:

```
eliza> Hello, how are you?
human> I feel really tired today.
eliza> Why do you feel really tired today?
human> I stayed up all night doing CS 135.
eliza> You stayed up all night doing CS 135?
human> They make us work too hard.
eliza> Can you be more specific?
human> Assignment 8 was a killer.
eliza> Tell me more about Assignment 8.
```

You can see how it wouldn't take a lot of processing to do this. In fact, we are going to do even less processing than Weizenbaum's program, which remembered past interactions in a certain way and could bring up phrases from them. Our program will treat every input from the human in isolation. To avoid dealing with the complexities of parsing strings, we will assume the input and output are lists of symbols:

```
> (eliza '(I feel really tired today))
'(Why do you feel really tired today)
```

We're not going to bother with punctuation, either. Since this is an enhancement, you can put the ability to handle strings with punctuation instead of lists of symbols into your implementation if you wish. (To get output that uses quote notation, select Details in the Choose Language dialog, and choose quasiquote.)

The key to writing an *eliza* procedure lies in patterns. The patterns we use in *eliza* allow the use of the question mark *?* to indicate a match for any one symbol, and the asterisk $*$ to indicate a match for zero or more symbols. For example:

'(I ? you) matches '(I love you) and '(I hate you), but not '(I really hate you).
'(I $*$ your ?) matches '(I like your style) and '(I really like your style), but not '(I really like your coding style).

We can talk about the parts of the text matched by the pattern; the asterisk in '(I $*$ your ?) matches '(really like) in the second example in the previous paragraph. Note that there are two different uses of the word "match": a pattern can match a text, and an asterisk or question mark (these are called "wildcards") in a pattern can match a sublist of a text.

What to do with these matches? We can create rules that specify an output that depends on matches. For instance, we could create the rule
'(I $*$ your ?) $\rightarrow$ '(Why do you 1 my 2)

which, when applied to the text '(I really like your style), produces the text '(Why do you really like my style).

So *eliza* is a program which tries a bunch of patterns, each of which is the left-hand side of a rule, to find a match; for the first match it finds, it applies the right-hand side (which we can call a "response") to create a text. Note that we can't use numbers in a response (because they refer to matches with the text) but we can use an asterisk or question mark; we can't use an asterisk or question mark in a pattern except as a wildcard. So we could have added the question mark at the end of the response in the example above.

A text is a list of symbols, as is a pattern and a response; a rule is a list of pairs, each pair containing a pattern and a response.

Here's how we suggest you start writing *eliza*.

First, write a function that compares two lists of symbols for equality. (This is basic review.) Then write the function *match-quest* which compares a pattern that might contain question marks (but no asterisks) to a text, and returns *true* if and only if there is a match.

Next, write the function *extract-quest*, which consumes a pattern without asterisks and a text, and produces a list of the matches. For example,

(*extract-quest* '(CS ? is ? fun) '(CS 135 is really fun))
$=>$ '((135) (really))

You are going to have to decide whether *extract-quest* returns *false* if the pattern does not match the text, or if it is only called in cases where there is a match. This decision affects not only how *extract-quest* is written, but other code developed after it.

Finally, write *match-star* and *extract-star*, which work like *match-quest* and *extract-quest*, but on patterns with no question marks. Test these thoroughly to make sure you understand.