CS 565 Out: August 28, 1997
Jens Palsberg Due: September 4, 1997

Homework 1:

Scheme Programming

Define and test the procedures outlined below. In doing the assignment, you may assume that the inputs to your functions have the correct type. This means that you need not check the inputs for validity. However, this also means that your function must behave gracefully (that is, not bomb) on all such inputs.

Your solution should consist of the required function definitions, including comments and any necessary auxiliary functions, and a suite of tests adequate to convince the grader that your solutions work correctly on all valid inputs.

You should turn both your solution and a transcript of your tests. The command `(transcript-on` *filename*`)` will make a transcript of your session on the given file. `(transcript-off)` ends transcription and closes the transcript file.

Note: You can "warm up" to doing the exercises by doing some of the Exercises 2.2.7 and 2.2.8 in *EOPL* (pp. 50–54). You can find solutions to these exercises on the homepage for CS 565, Fall 1996.

1.

Write a Scheme function `show-up-to-n` that takes as an argument a list of integers l and an integer n, and returns a list like the original, except that all elements that are greater than or equal to n are removed. For example,

```
(show-up-to-n '(4 6 3 -8 3 4) 5) ==> (4 3 -8 3 4)
(show-up-to-n '(3 4 6 3 2 2 4) 3) ==> (2 2)
```

2.

Write a Scheme procedure `double-add` that takes a list of integers l, and returns a pair where the first component is the sum of all negative integers in l

and the second component is the sum of all positive integers in l. Convention: the sum of *no* integers is 0. For example,

```
(double-add '(5 3 -2 4 -6)) ==> (-8 . 12)
(double-add '(1 2 3 4)) ==> (0 . 10)
(double-add '(1 -1 2 -2 3 -3)) ==> (-6 . 6)
```

3.

Write a Scheme function `add-all-pairs` that takes as arguments two lists of integers l and m, and returns a list which contains all sums of two integers where the first integer is from l and the second integer is from m. The order of the integers in the returned list is implementation dependent (any order is okay!). For example,

```
(add-all-pairs '(1 2) '(30 40)) ==> (31 32 41 42)
(add-all-pairs '(1 2) '()) ==> ()
```

4.

Write a Scheme procedure `split` that takes as arguments a list of integers l and an integer i, and returns a list where the first component is a list containing all elements of l which are less than or equal to i, and the rest are the elements of l which are greater than i. For example,

```
(split '(5 3 -2 4 -6) 0) ==> ((-2 -6) 5 3 4)
(split '(1 2 3 4) 1) ==> ((1) 2 3 4)
(split '(1 2 3 4) 5) ==> ((1 2 3 4))
```

5. Write a Scheme procedure `curry` that takes as argument a function f of two arguments, and returns a function which "takes the two arguments one at a time." In other words, the result of applying `curry` to a function f is a function g which when given an argument x returns another function h which, in turn, when given an argument y returns the result of applying f to both x and y. For example,

```
(((curry cons) 'a) '(3 4 5)) ==> (a 3 4 5)
(((curry split) '(5 3 -2 4 -6)) 0) ==> ((-2 -6) 5 3 4)
```

6. Write a Scheme procedure `tautology?` that takes as argument an integer n (where $n > 0$) and a boolean-valued function f which takes n boolean arguments "one at a time." The call `(tautology?` n f`)` returns `#t` if f returns always returns `#t`, and `#f` otherwise. For example,

```
(tautology? 1 (lambda (x) x)) ==> #f
(tautology? 2 (lambda (x) (lambda (y) #t))) ==> #t
(tautology? 2 (lambda (x) (lambda (y)
    (implies (implies (implies x y) x) x)))) ==> #t
;; where
(define implies (lambda (x y) (or (not x) y)))
```

7. Write a Scheme procedure `tautology-n?` that takes as argument an integer n (where $n > 0$) and a boolean-valued function f which takes n boolean arguments "at the same time." The call `(tautology-n?` n f`)` returns `#t` if f returns always returns `#t`, and `#f` otherwise. For example,

```
(tautology-n? 1 (lambda (x) x)) ==> #f
(tautology-n? 2 (lambda (x y) #t)) ==> #t
(tautology-n? 2 (lambda (x y)
    (implies (implies (implies x y) x) x))) ==> #t
;; where
(define implies (lambda (x y) (or (not x) y)))
```

8. Write a Scheme procedure `sieve` which takes as argument a number n (where $n > 1$) and returns a list with all prime numbers between 1 and n, possibly including n. The implementation should use the sieve of Eratosthenes. For example,

```
(sieve 5) ==> (2 3 5)
(sieve 12) ==> (2 3 5 7 11)
```