Assignment 09
Due: Tuesday, July 25th, 2017 at 4pm
NOTE the different day and time.

• You must provide the data definition and template in your solutions only when the question specifically indicates they are required for compound data types described in the question. If you create any additional data types that are beyond the question description, your program file should include a data definition and a template for each additional data type.
• If you include a template in your solution, the template should appear as comments.
• You may want to include defined constants to help reduce the writing for the examples and tests
• Unless otherwise indicated by the question you may only use the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment. A list of functions described in each module of the lecture slides may be found at https://www.student.cs.uwaterloo.ca/~cs115/built_in.
• Use the design recipe when writing functions (and helper functions) from scratch.
• Download the interface file from the course Web page to ensure that all function names are spelled correctly, and each function has the correct number and order of parameters.
• Read each question carefully for restrictions.
• Test data for all questions will always meet the stated assumptions for consumed values.
• Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include references to the parameter names of your functions.
• The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
• Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
• Check Markus and your basic test results to ensure that your files were properly submitted. In most cases, solutions that do not pass the basic tests will not receive any correctness marks.
• Any string or symbol values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks.
• Read the course Web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the Style Guide. Your solutions should be placed in files a09qY.rkt, where Y is a value from 1 to 5.

Language level: Intermediate Student with Lambda
Coverage: Modules 9 and 10

For questions 1, 2, and 3, you are not allowed to use recursion. Your solutions must use abstract list functions. All constants and helper functions must be local definitions. For full marks, your solution should be efficient and not repeat any work.

1. Write a Racket function substitute that consumes two strings, oldst and newst, and a list of strings, slst, and produces a list of the same length as slst but with all occurrences of oldst replaced by newst.

For example:
(substitute "old" "new" (list "this" "old" "new" "wow" "old" "cold"))
=> (list "this" "new" "new" "wow" "new" "cold"
2. Use the following structure and data definition for this question.

```racket
(define-struct daytime (hours minutes))
;; A DayTime is a (make-daytime Nat Nat)
;; requires 6 <= hours <= 18 and minutes < 60
```

Write a Racket function `bus-schedule` that consumes a non-zero natural number, interval, and produces the bus schedule which is a list of DayTime structure. The bus schedule starts at the sixth hour (6am), increases by the `interval`, and ends at eighteenth hour (6pm) inclusive, i.e., the difference between two consecutive DayTime in the produced list is the `interval` value.

For example:

```racket
(bus-schedule 90) =>
(list (make-daytime 6 0) (make-daytime 7 30) (make-daytime 9 0)
      (make-daytime 10 30) (make-daytime 12 0) (make-daytime 13 30)
      (make-daytime 15 0) (make-daytime 16 30) (make-daytime 18 0))

(bus-schedule 50) =>
(list (make-daytime 6 0) (make-daytime 6 50) (make-daytime 7 40)
      (make-daytime 8 30) (make-daytime 9 20) (make-daytime 10 10)
      (make-daytime 11 0) (make-daytime 11 50) (make-daytime 12 40)
      (make-daytime 13 30) (make-daytime 14 20) (make-daytime 15 10)
      (make-daytime 16 0) (make-daytime 16 50) (make-daytime 17 40))
```

3. Use the following structure and data definition to complete this question:

```racket
(define-struct vehicle (make model))
;; A Vehicle is a (make-vehicle Str Str)
```

Write a Racket function `list-models` that consumes a string `make`, representing the make of a vehicle, and a list of Vehicle structure `vehicles`, and produces a list of all models of the vehicle make.

For example:

```racket
(list-models "chevy"
  (list (make-vehicle "chevy" "cavalier") (make-vehicle "chevy" "equinox")
       (make-vehicle "kia" "sorento") (make-vehicle "toyota" "camry")))
=> (list "cavalier" "equinox")

(list-models "toyota"
  (list (make-vehicle "chevy" "cavalier") (make-vehicle "chevy" "equinox")
       (make-vehicle "kia" "sorento") (make-vehicle "toyota" "camry")))
=> (list "camry")
4. Use the following structure and data definition to complete this question:

```
(define-struct ainode (op args))

;; An Arithmetic expression internal node (AINode)
;; is a (make-ainode (anyof '* '+) (listof AExp))
;; requires args is a non-empty list

;; An Arithmetic Expression (AExp) is one of:
;; * a Num
;; * an AINode
```

Recall the substitution rules in Racket from Module 1 that are applied when completing a full trace of a Racket expression. Assume that you are tracing an arithmetic expression that only contains non-negative numbers, and the mathematical functions + and *. Here is a sample trace:

```
(+ 5 (* (+ 1 2 3) 2 (* 4 1)) 7)
=> (+ 5 (* 6 2 (* 4 1)) 7)
=> (+ 5 (* 6 2 4) 7)
=> (+ 5 48 7)
=> 60
```

The substitution rules that are used in this kind of trace are:
- A value (such as a number) cannot be further simplified.
- For a built-in function application, use mathematics rules.
- First evaluate the arguments, and then apply the function to the resulting values.
- When there is a choice among two or more substitutions, take the leftmost one.

Each of the lines in the trace above is an example of an arithmetic expression in Racket. Any arithmetic expression in Racket can be represented by an AExp.

Write a Racket function `step` that consumes an AExp, `ex` and produces an AExp that represents the next step in a trace of `ex`. If `ex` is a number, then there are no more steps in the trace. In this case the function produces the symbol 'Done. Simplifying an argument such as `(+ 1 2 3)`, where all of the arguments following the mathematical operator are numbers, is done in just one step.

For example:
- `(step (+ 5 (* (+ 1 2 3) 2 (* 4 1)) 7)) => 'Done`
- `(step (make-ainode '* (list 0))) => 0`
- `(step (make-ainode '+ (list 1 (make-ainode '+ (list 1 1)) 3 4 5))) => (make-ainode '+ (list 1 2 3 4 5))`
In this question, you will deal with an extension of the alternate implementation of arithmetic expressions discussed in Module 10, in which a variable 'x is introduced as additional base case for arithmetic expressions. The following new type (VAExp) is introduced:

```scheme
;; A Variable Arithmetic Expression (VAExp) is one of:
;; * the symbol 'x
;; * a number
;; * (cons (anyof '+ '*)) (listof VAExp))
```

Write the Racket function `eval-x` that consumes `ex` (a VAExp, as defined above) and `x-val` (a Num), and produces the simplified value of `ex`, when the symbol 'x is replaced with `x-val`.

For example,
- `(eval-x 4 -1)` => 4
- `(eval-x 'x 3)` => 3
- `(eval-x (list '* 'x (list '+ 'x 3)) -1)` => -2
- `(eval-x (list '+ (list '* (list '* 'x 4) 5) 0) => 6`

Recall that `(list '+)` evaluates to 0 and `(list '*)` evaluates to 1.