Date Due: Monday March 8th, 1:00 PM

CS 116
Midterm 2
Winter 2010

Instructors: Terry Anderson, Lori Case, Leila Chinaei, Sandy Graham, Tian Kou

Exam Instructions

- Please keep in mind that this midterm is due on **Monday** March 8th at **1:00 PM** (this is different from the usual assignment due date/time).
- This is to be treated as a midterm. **You must complete all problems without help from anyone else. Collaboration with other students will not be tolerated.**
- In your solutions, you must include all components of the design recipe (except for Question 4). However, only some of these components will be marked for each question.
- 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.
- Course personnel will not answer any questions about the problems except to clarify the problem statement. Most clarifications will be posted to the announcement section of the course website.
- Solutions to all questions will be submitted electronically. **No late submissions will be accepted.** It is very important that you submit early and often.
- Your solutions must be placed in the interface files provided on the course website. Do not change any of the filenames.
- There will be no public tests available for these problems.
- Each of the six questions will be equally weighted in the marking scheme.
- This exam has 8 pages (including the cover page).
- Relax! Read this instruction as often as needed.

Language Level for questions 1, 2, and 3: Advanced Student Scheme

1. Create a Scheme function `vending-machine` that simulates the actions of a vending machine. The program contains three state variables: `coin-box` (representing the total amount of money inserted into the vending machine), `num-chips` (representing the number of bags of chips in the machine) and `num-bars` (representing the number of chocolate bars in the machine). The program also contains constants that indicate the price of chips and chocolate bars. The function `vending-machine` will consume a number equal to the amount of a coin (one of `0.05`, `0.10`, `0.25`, `1.00`, `2.00`) or a symbol indicating a purchase (one of `'chips` or `'bar`). The function will produce the following:

   - If the value entered is a number, nothing is produced, but the `coin-box` is updated accordingly.
   - If the value entered is a symbol representing a food item with an inventory of 0, the function produces the string `"Sold out"` regardless of how much money is in the `coin-box`. (The `coin-box` will be unaffected.)
   - If the value entered is a symbol and the `coin-box` contains enough money for the purchase matching the symbol, the function will produce the change owed for the purchase, the `coin-box` should be reset to 0 and the appropriate food state variable is reduced by 1.
   - If the value entered is a symbol and the coin-box does **not** contain enough money, the function will produce the string `"Insufficient funds"` . (The `coin-box` will be unaffected either way.)

   For example, suppose the following sequence of function calls occurred:

   ```
   (set! num-bars 5)
   (set! coin-box 0)
   (vending-machine 2.00)
   (vending-machine 'bar)
   ```

   Then the final function call would produce `0.5`, value of `coin-box` would be `0`, and the value of `num-bars` would be `4`.

   You may assume that all input to the function will be valid (that is, all the numbers will be greater than 0, and any symbol will be either `'chips` or `'bar`.)
   Complete your solution without adding any extra state variables. You may use local variables in the body of your function.

2. This question uses the structures **athlete** and **winners**, defined below:

(define-struct athlete (name country placement))
;; An **athlete** is a structure *(make-athlete n c p)* where
;; *n* is a string (athlete's name), *c* is a string (country the athlete represents)
;; *p* is a natural number (place the athlete finished in the competition) or
;; a symbol (eg. 'DNF indicating the athlete did not finish the competition or
;; 'WD indicating the athlete withdrew from the competition before it started)
(Note this is a slightly different definition than the one used on midterm one)

(define-struct winners (description gold silver bronze))
;; A **winners** is a structure *(make-winners d g s b)*, where
;; *d* is a string (description or name of the athletic competition)
;; *g,s,b* are athletes (*g*, for gold, is the athlete who placed first,
;; *s*, for silver, is the athlete who placed second, and *b*, for bronze,
;; is the athlete who placed third in the competition).

Create a Scheme function called `remove-cheater` that consumes a string (representing the name of a cheater), a `winners` structure (representing the medalists in the event), and an athlete (representing the fourth place finisher in the event in which the `winners` competed.) If the name of the cheater is one of the medalists, the `placement` of that athlete will be changed to `'DQ`, and everyone else will move up one place in the standings. For example, if the gold medalist is the cheater, then the `placement` of the former gold medalist changes to `'DQ`, the silver medalist moves to the gold position (and their `placement` changes to 1), the bronze medalist moves to the silver position (and their `placement` changes to 2), and the fourth place finisher moves to the bronze position (and their `placement` changes to 3). The `gold`, `silver`, and `bronze` fields of the `winners` structure should also be updated appropriately.

If the name of the cheater is neither a medalist nor the fourth place finisher, nothing will change. In this case the function should produce the string `"No change"`. In all other cases the function should produce `(void)`.

You may not use the `make-winners` or `make-athlete` functions in the body of the `remove-cheater` function (or any helper functions you write). However these may be used within your test cases.

3. For this question, you will write a Scheme function `has-substring?`. This function consumes two strings, `text` and `pattern`, and will produce `true` if `pattern` is a substring of `text` and `false` otherwise.

We can immediately determine that `pattern` is a substring of `text` if the two strings are equal, and we can immediately determine that `pattern` is **not** a substring of `text` if:

- `text` is shorter than `pattern`, or
- `text` is the same length as `pattern` but the two strings are not equal.

Otherwise we consider the following cases. Let `left` be the first half of `text` and let `right` be the second half of `text` (if the length of `text` is odd, then make `left` one character shorter than `right`). We can determine that `pattern` is a substring of `text` if:

- `pattern` is a substring of `left`, or
- `pattern` is a substring of `right`, or
- `pattern` is a substring of `text` such that `text` has its first character removed, or
- `pattern` is a substring of `text` such that `text` has its last character removed.

Note that the last two cases are necessary because it is possible for `pattern` to be a substring of `text` but be a substring of neither `left` nor `right` (for example, when `text` is `"telephones"` and `pattern` is `"leph"`).

Your solution **must use generative recursion** to implement the algorithm **exactly as it is described in this question**, even if you can think of an alternate way to solve the problem.

Examples:

```
(has-substring? "abc" "abc") => true
(has-substring? "telephones" "leph") => true
(has-substring? "quick" "xy") => false
(has-substring? "011000101101" "111") => false
```

4. Consider the four functions **`reverse-search`**, **`sum-of-divisors`**, **`grow-duplicates`**, and **`mystery-sort`**. For *each* of these functions, state:

   - The **best case** running time of the function along with a brief justification, and
   - The **worst case** running time of the function along with a brief justification.

   If the best case running time and the worst case running time for a function are *different*, you must also provide:

   - An input value on which the function achieves its best case running time, and
   - An input value on which the function achieves its worst case running time.

   When giving the best case running time of a function, you are not allowed to select the size of the input. Specifically, do not make statements such as "the best case is when the list is empty" or "the best case is when the list has length one." The size of the input, n, is always considered to be an arbitrary value.

   The contract for each function (and local helper function) appears in bold. We have omitted the purpose, examples, and test cases for all functions; this is to encourage you to thoroughly read and trace the code itself to understand exactly how the function works. The ability to trace a function on a variety of input values is of great benefit when determining its best case and worst case running times.

   You may make the following assumptions for this question:

   - All of the running times will be one of the following:
     - Constant (also expressed as $O(1)$)
     - Linear (also expressed as $O(n)$)
     - Quadratic (also expressed as $O(n^2)$)
     - Exponential (also expressed as $O(2^n)$)
   - The function `reverse` has linear running time

```
; reverse-search: (listof any) any → boolean
(define (reverse-search lst target)
   (cond
      [(empty? lst) false]
      [(equal? (first lst) target) true]
      [else (reverse-search (reverse (rest lst)) target)]))
```

```
; sum-of-divisors: nat → nat
(define (sum-of-divisors n)
   (foldr + 0
      (filter (lambda (y) (zero? (remainder n y)))
          (build-list n (lambda (x) (add1 x))))))
```

```
; grow-duplicates: (listof X) → (listof X)
(define (grow-duplicates lst)
   (cond
      [(empty? lst) empty]
      [(= 1 (length lst)) empty]
      [(empty? (filter (lambda (x) (equal? x (first lst))) (rest lst)))
               (grow-duplicates (rest lst))]
      [else (append (grow-duplicates (rest lst))
                    (list (first lst)) (grow-duplicates (rest lst)))]))
```

```
; mystery-sort: (listof X)[nonempty] → (listof X)[nonempty]
; The elements of type X must be comparable with < and >
(define (mystery-sort lst)
  (local
      ; partition: (listof X) (listof (listof X)) (listof X)[nonempty]
      ;                                → (listof (listof X))[nonempty]
      ; The elements of type X must be comparable with < and >
      [(define (partition lst0 list-of-lists next-list)
         (cond
            [(empty? lst0)
               (reverse (cons (reverse next-list) list-of-lists))]
            [(> (first lst0) (first next-list))
               (partition (rest lst0) list-of-lists
                  (cons (first lst0) next-list))]
            [else (partition (rest lst0)
               (cons (reverse next-list) list-of-lists)
                  (list (first lst0)))]))
      ; merge: (listof X) (listof X) → (listof X)
      ; The elements of type X must be comparable with < and >
      (define (merge lst1 lst2)
         (cond
            [(empty? lst1) lst2]
            [(empty? lst2) lst1]
            [(<= (first lst1) (first lst2))
               (cons (first lst1) (merge (rest lst1) lst2))]
            [else (cons (first lst2) (merge lst1 (rest lst2)))]))]
  (foldr merge empty (partition (rest lst) empty (list (first lst))))))
```

5. Create a Python function `identify_triangle` that prompts the user to enter three positive integer values in **ascending** order: `a`, `b`, and `c`. These values represent the lengths of the sides of a triangle. If the length of the longest side is greater than or equal to the sum of the lengths of the two smaller sides, the triangle is considered invalid. A valid triangle can be categorized as **one** of the following based on the lengths of its sides:

   - an **equilateral** triangle has three equal sides
   - an **isosceles** triangle has two equal sides
   - a **scalene** triangle has no equal sides

   A triangle can also be categorized as **one** of the following relating to the interior angles:

   - a **right-angled** triangle has sides where $c^2 = a^2 + b^2$ where a, b, and c are the lengths of the sides of the triangle
   - an **acute** triangle has sides where $c^2 < a^2 + b^2$ where a, b, and c are the lengths of the sides of the triangle
   - an **obtuse** triangle has sides where $c^2 > a^2 + b^2$ where a, b, and c are the lengths of the sides of the triangle

   Your function should print `invalid` if the triangle is invalid. Otherwise, it should print the category of the triangle based on its side lengths (`equilateral`, `isosceles`, or `scalene`) followed by the category of the triangle based on its interior angles (`right-angled`, `acute`, or `obtuse`).

   Your solution should use the prompts provided in the starter file. Also, whenever you are working with strings in your programs, it is very important to be precise. All strings that are printed should be lower case and without any extra whitespace. If there is more than one string being printed, each string should appear on its own line. Do not print out any extra blank lines.

   Three sample runs of the program are shown below. Note the use of bold to indicate the values entered by the user while the program is executing. You may assume that the user will always enter positive integer values.

   ```
   Enter the value of a: 3
   Enter the value of b: 3
   Enter the value of c: 10
   invalid
   ```

   (Sample runs 2 and 3 appear on the next page)

```
Enter the value of a: 5
Enter the value of b: 5
Enter the value of c: 8
isosceles
obtuse
```

```
Enter the value of a: 3
Enter the value of b: 4
Enter the value of c: 5
scalene
right-angled
```

6. Decimal numbers, or base 10 numbers, are written using 10 possible digits. Numbers can be represented with any base. For example binary numbers, or base 2 numbers, are written using just two digits: 0 and 1. The binary number 1101 is equivalent to the decimal number 13. The number 200 in base 5 is 1300. Create a Python function called `convert_base` that will consume a non-negative integer and an integer between 2 and 9 inclusive, and produce a string representing the number in the new base. Note that 0 is the same number in any base.

   To convert a number to a different base, you need to continuously divide by the base until you reach 0 and record the remainder each time. Then write the remainders in reverse order.

   Your solution must use recursion; it may not use loops.

   Example 1: Converting 13 to base 2

   13/2 = 6 remainder 1
   6/2 = 3 remainder 0
   3/2 = 1 remainder 1
   1/2 = 0 remainder 1
   Looking at the remainders in reverse, this produces 1101 in base 2, and this is the equivalent of 13 in base 10.

   Example 2: Converting 500 to base 7

   500/7 = 71 remainder 3
   71/7 = 10 remainder 1
   10/7 = 1 remainder 3
   1/7 = 0 remainder 1
   Looking at the remainders in reverse, this produces 1313 in base 7, and this is the equivalent of 500 in base 10.