CMSC 198Q, Midterm 1
SOLUTION

Summer 2019

NAME:_____

Question	Points
1	10
2	10
3	15
4	15
5	20
Total:	70

This test is open-book, open-notes, but you may not use any computing device other than your brain and may not communicate with anyone. You have 60 minutes to complete the test.

The phrase "design a program" or "design a function" means follow the steps of the design recipe. Unless specifically asked for, you do not need to provide intermediate products like templates or stubs, though they may be useful to help you construct correct solutions.

You may use any of the data definitions given to you within this exam and do not need to repeat their definitions.

Unless specifically instructed otherwise, you may use any built-in BSL functions or data types.

When writing tests, you may use a shorthand for writing check-expects by drawing an arrow between two expressions to mean you expect the first to evaluate to same result as the second. For example, you may write `(add1 3)` $\rightarrow$ `4` instead of `(check-expect (add1 3) 4)`.

Problem 1 (10 points). For the following program, write out each step of computation. At each step, underline the expression being simplified. Label each step as being "arithmetic" (meaning any built-in operation), "conditional", "plug" (for plugging in an argument for a function parameter), or "constant" for replacing a constant with its value.

```
(define R 2)
(define (q s) (< R (string-length s)))
(cond [(= 1 2) (q "fred")]
      [else (q "wilma")])
```

SOLUTION:

```
(cond [(= 1 2) (q "fred")] [else (q "wilma")]) -->[arith]
       ^^^^^^^

(cond [#false (q "fred")] [else (q "wilma")]) -->[cond]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

(cond [else (q "wilma")])                     -->[cond]
^^^^^^^^^^^^^^^^^^^^^^^^

(q "wilma")                                   -->[plug]
^^^^^^^^^^^

(< R (string-length "wilma"))                 -->[const]
   ^

(< 2 (string-length "wilma"))                 -->[arith]
     ^^^^^^^^^^^^^^^^^^^^^^^

(< 2 5)                                       -->[arith]
^^^^^^^

#true
```

Problem 2 (10 points). For the following structure definition, list the names of every function it creates. For each function, classify it as being either a constructor, accessor, or predicate.

```
(define-struct up (side vote chuck))
```

SOLUTION:

```
- make-up    : Constructor
- up-side    : Accessor
- up-vote    : Accessor
- up-chuck   : Accessor
- up?        : Predicate
```

```
(define U (make-up "exam" #false (make-up 2019 #true "red")))
```

Using only accessor functions and `U`, write an expression that produces `2019`.
SOLUTION:

```
(up-side (up-chuck U))
```

Problem 3 (15 points). You've been hired by Wheel of Fortune, a popular game show where contestants try to reveal a hidden word or phrase by guessing letters. So for example, if the hidden word is "HELLO," they start knowing there are five letters: _ _ _ _ _. Suppose they guess "L," then the two occurrences are revealed: _ _ L L _. If they the guess "H," they'd see: H _ L L _, and so on.

You start by focusing on the representation of individual (potentially hidden) letters and settle on the following data definition (here `1String` just means a string of length 1):

```
;; A Letter is one of:
;; - (make-hidden 1String)
;; - 1String
;; Interp: (make-hidden s) means s has not yet been guessed and is
;; being hidden, otherwise the letter has been guessed and revealed.
(define-struct hidden (letter))
```

So `(make-hidden "L")` represents the letter `"L"` being hidden in a word or phrase. Once guessed, it would become simply `"L"`.

Desing a function `guess` which takes a `Letter` and a `1String` and reveals the hidden letter if the guess is correct and otherwise leaves it alone if it's incorrect or the letter is already revealed.

`SOLUTION:`

```
;; Letter 1String -> Letter
(check-expect (guess "a" "b") "a")
(check-expect (guess (make-hidden "a") "b") (make-hidden "a"))
(check-expect (guess (make-hidden "a") "a") "a")
(define (guess l s)
  (cond [(hidden? l)
         (cond [(string=? s (hidden-letter l)) s]
               [else l])]
        [(string? l) l]))
```

Problem 4 (15 points). Points in three-dimensional space are a lot like their two-dimensional counterparts, just with one more dimension. So rather than pairs of numbers (x, y), 3D points are determined by triples (x, y, z). Similarly, the distance of a 3D point to the origin is a straightforward extension of the formula for calculating the distance of a 2D point to the origin:

$$\sqrt{(x^2 + y^2 + z^2)}$$

And the point $(2, 3, 6)$ has a distance of 7 to the origin.

Design a data representation for 3D points and a function for computing the distance of a 3D point to the origin.

SOLUTION:

```
;; A 3D is a (make-3d Number Number Number)
(define-struct 3d (x y z))

;; 3D -> Number
;; Compute distance to origin
(check-expect (dist (make-3d 2 3 6)) 7)
(define (dist p)
  (sqrt (+ (sqr (3d-x p)) (sqr (3d-y p)) (sqr (3d-z p)))))
```

Problem 5 (20 points). After completing the simple line text editor you developed in lab 4, you decide to extend your design so that in addition to representing text with a cursor somewhere within it, you can also represent text where some portion of it has been "selected" (for example if the user were to click and drag a cursor across some part of the text).

```
;; A Line is one of:
;; - (make-txt String String)
;; - (make-sel String String String)
;; Interpretation: (make-txt string1 string2) is a line of text consisting
;; of string1 and string2 with a cursor placed between them.
;; (make-sel string1 string2 string3) is a line of text consisting of
;; string1, string2, and string3, but string2 is currently selected.
(define-struct txt (left right))
(define-struct sel (left selected right))
```

So for example, if you want to represent the text "Hello" with "ll" selected, it would be

```
(make-sel "He" "ll" "o")
```

Design a function `select-all` that takes a line and selects all available text in the line, regardless of where the cursor is or what may currently be selected.

SOLUTION:

```
;; Line -> Line
;; Select all of the text in the line
(check-expect (line-select (make-txt "a" "b")) (make-sel "" "ab" ""))
(check-expect (line-select (make-sel "a" "b" "c")) (make-sel "" "abc" ""))
(define (line-select-all l)
  (cond [(txt? l) (make-sel "" (string-append (txt-left l) (txt-right l)) "")]
        [(sel? l)
         (make-sel ""
                   (string-append (sel-left l)
                                  (sel-selected l)
                                  (sel-right l)) "")]))
```