CSU2500 Exam 2 HONORS SUPPLEMENT – Fall 2010

Name: _____

Student Id (last 4 digits): _____

- This supplement to Exam 2 is intended for students enrolled in the Honors section of 2500.
- See the instructions on the regular exam.

Problem	Points	/out of
1		/ 16
2		/ 13
3		/ 8
Total		/ 37

Good luck!

Problem 1 The word "*parity*" is sometimes used to refer to how many ones are in a number when it is represented in base 2. For example, the numbers 3 and 5, which when written in base 2 are 11 and 101, both have *even parity*, while the numbers 2 and 7, written in base 2 as 10 and 111, both have *odd parity*. 16 POINTS

We can generalize the notion of parity to arbitrary lists and predicates. Design the function, `parity`, that accepts a predicate and a `[Listof X]` and determines if list's *parity* is even or odd with respect to the predicate. When a list has an *odd* parity, the function should produce `true`, and when it has an even parity, it should produce `false`.

For example:

```
(define (one? n) (= n 1))

(check-expect (parity one? '(1 1)) false)
(check-expect (parity one? '(1 0)) true)
(check-expect (parity one? '(1 1 1)) true)
(check-expect (parity symbol? '(i b 4 e)) true)
```

Your Tasks:

A) Design `parity` using only structural recursion (i.e., no loop function).

B) Write `parity` using a single call to `foldr`.

[Here is some more space for the previous problem.]

Problem 2 All semester students have been asking us to use objects, so we've decided to show you some on the exam. How would we represent objects in a functional language like ISL-λ? As functions of course! For this problem you will implement a "*class*" of `Point` objects. A `Point` is an object-oriented (OO) representation of coordinates (similar to `Posns`), though you don't need to know *anything* about objects to do this problem; just pay careful attention to the description and the examples.

13 POINTS

Design a function `new-point` that consumes two numbers (an x- and a y-coordinate) and produces a `Point`.

```
;; new-point : Number Number -> Point
```

A `Point` is a function that responds to *messages*. A message is sent by applying a `Point` to a `Symbol` that matches the message's name. The object reacts by producing a value, which is frequently a "method," that is, a function that will carry out some task on behalf of the object.

Here are the *contracts* of the messages your `Point` representation must support:

Message Name	Message Result Contract
`'x`	`Number`
`'y`	`Number`
`'move`	`[Number Number -> Point]`
`'same`	`[Point -> Boolean]`

Sending a `Point` the message `'x` (in other words, applying a `Point` to the symbol `'x`) returns a number that represents the x-coordinate of the point (the first argument to `new-point`); sending `'y` returns the y-coordinate. Sending a `Point` the message `'move` returns a function that consumes x and y offsets and constructs a new point with x and y moved by the given amounts. Sending a `Point` the message `'same` returns a function that when applied to another `Point` determines if the points have the same x- and y-coordinates.

Hint: The next page contains some examples/tests to further clarify the details.

Task: Design `new-point`.

```
;; Example Points...
(define p0 (new-point 0 0))
(define p1 (new-point 3 4))

;; Tests for each 'message'
(check-expect (p1 'y) 4)
(check-expect (* (p1 'x) (p1 'y)) 12)

(check-expect (((p1 'move) 14 13) 'x) 17)

(check-expect ((p1 'same) p0) false)
(check-expect ((p1 'same) p1) true)
(check-expect ((((p1 'move) -3 -4) 'same) p0) true)
```

[Here is some more space for the previous problem.]

Problem 3 Your startup company hires a Northeastern co-op who spends six months coding up a very large and complex library for doing 2D computational geometry. On the student's last day at the company, disaster strikes: a demo of his library reveals that he has used algorithms for his libary that all rely on *rectangular coordinates* to represent 2D points on a plane using `posn`s, while your company's applications, databases, *etc.* all represent data using *polar coordinates.* No one remembered to tell him that your company's code all represents points using `polar` structs:

8 POINTS

```
;; A PolarPt is: (make-polar Number Number)
(define-struct polar (r theta))
```

Six months of work... down the drain.

Or is it? In a fit of inspiration, you sit down and wish up a function,

```
rect-fun->polar-fun
```

The input to this function is another function, f, that consumes a `posn` representing a point on the plane in $(x,\ y)$ rectangular coordinates and produces some value. The output of your function is a polar-coordinate version of f, that is, a function g that consumes a `PolarPt` representing a point on the plane in $(r,\ \theta)$ polar coordinates, and produces a result equivalent to what f would when applied to $(x,\ y)$. (Recall that a point which is at $(r,\ \theta)$ in polar coordinates is at $(r\cos\theta,\ r\sin\theta)$ in rectangular coordinates.)

If only you had this function, you could use it to convert all the functions in the student's library that consume rectangular-coordinate points to equivalent functions that consume polar-coordinate points—which means (1) that they then could be used by all the programmers at your company, (2) your poor co-op wouldn't have to commit ritual suicide, and (3) your boss would double your stock-option grant in gratitude. Not bad, for a couple of lines of code.

Stop wishing and design `rect-fun->polar-fun`.

Here is an example of a function that expects rectangular coordinates and a function constructed with `rect-fun->polar-fun` that does the same for polar coordinates. (You can consider this the test for your function, so no need to write more `check-expect`s.)

```
;; Posn -> Number
;; Compute distance to origin of a rectangular point.
(define (rect-dist p)
  (sqrt (+ (sqr (posn-x p))
           (sqr (posn-y p)))))

;; PolarPt -> Number
;; Compute distance to origin of a polar point.
(define polar-dist
  (rect-fun->polar-fun rect-dist))

(check-expect (polar-dist (make-polar 1 0)) 1)
(check-expect (polar-dist (make-polar 5 13)) 5)
```

[Here is some more space for the previous problem.]