UBC CPSC 110 - 2009 Winter T1 Final Exam

December 19th 2009

Please read this entire first page now.

Name: ____

UBC ID #: ____

Please do not open this exam until we ask you to do so.

Please write your answers neatly and only on the front side of each page.

Some of the problems on this exam ask for written answers, rather than code, numbers or letters. The best answers to such questions are short, concrete and to-the-point. Sometimes a small example program can help to make the answer clear. Do not attempt to write 'cover all the bases' answers to such questions – overly long answers will receive less credit.

As you write this example, do not worry about details like the exact name of big-bang handler options, or the order of arguments to big-bang handlers. For details like that just use a reasonable name and order of arguments. Points will not be deducted for details like that.

P1 - Language Syntax		/ 5
P2 - Designing World Programs		/ 15
P3 - Representing Information as Data		/ 15
P4 - Data Definitions, Examples, Tests and Templates		/ 20
P5 - Designing Functions		/ 20
P6 - Functional Abstraction		/ 10
P7 - Using Standard Iteration Functions		/ 5
P8 - Accumulators		/ 10
E1 - Extra Credit 1		/ 3
E2 - Extra Credit 2		/ 7
TOTAL		

Problem 1 - Language Syntax (5 points)

Some versions of Scheme have a construct called let-cc. The syntax of let-cc can be specified within the framework of ASL as follows:

```
<exp> = (let-cc <var> <exp>)
```

Which of the following are syntactically legal? (circle your answer)

```
(let-cc (fred) (wilma))      [LEGAL]   [ILLEGAL]

(let-cc a b)                 [LEGAL]   [ILLEGAL]

(let-cc 1 2)                 [LEGAL]   [ILLEGAL]
```

Problem 2 - Designing World Programs (15 points)

You are asked to build a 'flight simulation' that works as follows:

There is a window 400 pixels high and 800 pixels wide.

When the program starts the image of a plane is at the upper left. As time goes by the plane glides down and to the right until it touches the ground and stops.

You immediately realize that you will design a world program using `big-bang`, and that the first step is to do a domain analysis to identify information in the problem domain that must be represented using data in the program.

Write constant and graphical constant definitions as well as any needed structure definitions and a data definition for the WorldState. We have given you a rough image of a plane to use.

```
;; Constants:
```

```
;; Graphical constants:

(define PLANE-IMAGE (text ">>" 20 "black"))
```

```
;; Data definitions:
```

Now design the rest of the world program using **`big-bang`**. The **`main`** function that runs the animation should come first. (You may assume you are using ASL, so your main function is not required to consume arguments.) Note that in this program the simulation stops when the plane touches the ground -- the plane doesn't bounce! Use the **`stop-when`** clause of big-bang to stop the simulation then.

(this space is provided if you need it for problem 2)

Problem 3 - Representing Information as Data (15 points)

Each of the following partial problem descriptions describes some information in the problem domain that must be represented as data in the program. For each case, write an appropriate data definition. If necessary, include appropriate structure definitions.

NOTE: These partial problem descriptions in parts A, B and C are all separate -- do not consider them as being different parts of a larger system. Your analysis should only consider the fragment of the problem description you are given in each part.

(A) The system must keep track of the total number of records.

(B) For each traffic light keep track of its current color (one of "R" "Y" "G"); and keep track of all the traffic lights.

(C) For a book, record the title, author(s), isbn number, and whether it is checked out.

(D) For each parcel of land, a record must be kept of the sub-parcels it is divided into, and the sub-parcels each of those are divided into and so on.

Problem 4 - Data Definitions, Examples, Tests and Templates (20 points)

(Part A)

You are given the following data definition:

```
(define-struct foo (a b))
;; A Snarfle (s) is one of:
;;  - false
;;  - (make-foo a b) where
;;    a is a Number
;;    b is a String
```

Show three examples of a Snarfle.

Show the template for a function that operates on a Snarfle.

What minimum number of tests do the data definition and template suggest for this function?

(Part B)

You are given the following data definition:

```
(define-struct reading (type val))
;; A Reading is (make-reading t v) where:
;;   t is one of "hand" "auto"
;;   v is a Number an Integer in [0..10]
;; interp. The value of a measurement and how it was taken.
```

Show three examples of a Reading.

Show the template for a function that operates on a Reading.

What minimum number of tests do the data definition and template suggest for this function?

(Part C)

```
;;
;; A PedSignal (ps) is one of: "W" "H"
;; interp. whether the sign shows 'Walk' or a 'Stop Hand'
;;
;; A TLColor (tl) is one of: "R" "Y" "G"
;; interp. whether the light is red, yellow or green
;;
```

Show the template for a function operating on **BOTH** a PedSignal and a TLColor.

Problem 5 - Designing Functions (20 points)

In this question you will be working with the following structure and data definitions:

```
A Name is a string

(define-struct person (name friends))
;;
;; A Person is (make-node n lop) where:
;;  - n is a Name
;;  - lop is a ListOfPerson
;;
;; A ListOfPerson is either:
;;  - empty
;;  - (cons p lop) where p is a Person and lop is a ListOfPerson
;;
```

(Part A)

On the data definitions above, draw an arrow from every reference to the data definition it references above.

- Label any arrows that are self-references with SR.
- Label any arrows that are mutual-references with MR.
- Label the remaining arrows with R.

(Part B)

Now consider writing the template for a function that operates on Persons. Assume the function is called fun-for-person.

Based on the number of reference arrows you identified, how many helper functions should you use? (We want the number of HELPER functions, so do not include fun-for-person itself in your answer.)

(Part C)

Write the templates for fun-for-person and its helpers. Name all the functions consistently (fun-for-person, fun-for-lop etc.)

(Part D)

Design a find-for-person function (and its helpers) that consumes a Name and a Person, and returns true if a Person with that name is reachable from the Person. Feel free to use abbreviations in the definitions of your test cases, as long as what you mean is unambiguous. In other words, you need to make it clear to us what test cases you would write, but you don't need to write them all out in their gory detail.

(this space is provided if you need it for problem 5 D)

(Part E)

If we assume that the only function needed by the rest of our program is find-for-person, then the definitions above can be grouped into a single definition for find-for-person, with local definitions of its helper functions.

Provide a concise but specific and complete summary of what you would do to define this improved version of find-for-person. Don't just write out the standard process, be specific about what would happen in this case. We want to know the specific ways the code would be improved. If you want, you can just write the complete new definition for find-for-person.

Problem 6 - Functional Abstraction (10 Points)

The following two functions have a lot in common.

```
;; lengths: (listof Image) -> (listof Number)
;; strings: (listof Number) -> (listof String)

(define (heights loi)
  (cond [(empty? loi) empty]
        [else (cons (image-height (first loi))
                    (heights (rest loi)))]))

(define (squares lon)
  (cond [(empty? lon) empty]
        [else (cons (sqr (first lon))
                    (squares (rest lon)))]))
```

Use functional abstraction to define a higher-order function that can be used to simplify the implementation of heights and squares.

Reimplement heights and squares to use the new abstract function. You do not need to repeat the contracts or write tests.

Write a suitably abstract contract for your new abstract function.

Problem 7 - Using standard iteration functions (5 points)

What is the value of each of the following expressions?

```
(map list (list 1 2 3))
```

```
(filter even? (list 1 2 3 4 5 6))
```

```
(filter odd? (build-list 8 identity))
```

```
(foldr * 1 (list 1 2 3 4 5))
```

Problem 7 - Accumulators (10 points):

<u>Design</u> an accumulator-style version of the function make-palindrome, which accepts a non-empty list of one-character strings and constructs a palindrome by mirroring the list around the last item. For example:

```
(make-palindrome (list "a" "b")) --> (list "a" "b" "a"))

(make-palindrome (list "i" "p" "r" "e" "f")) -->
  (list "i" "p" "r" "e" "f" "e" "r" "p" "i")
```

You may assume that make-palindrome is never called with the empty list.

Extra Credit 1 (3 points):

Design an accumulator-style version of the fibonnacci function, or fib as it is usually called. The fibonnacci numbers are specified as follows:

```
(fib 0)  --> 0
(fib 1)  --> 1
(fib n)  --> (+ (fib (- n 1)) (fib (- n 2)))
```

Work according to the process described in the book and be sure to clearly document the accumlator invariants. You do not need to write test cases.

HINTS:

- Use TWO accumulators for this function.
- You may find it easier to design the function if you first assume it will never be called with n < 2, and then, once you have that designed properly, make a small change to it to handle the n=0 and n=1 cases.

Extra Credit 2 (7 points):

Warning, this problem is very hard! Do not start it until you have completed the rest of the final. It is likely to require a lot of thinking, and then writing out about 25 lines of code.

Alan has been working especially hard on his tic-tac-toe lab. He's figured out how to make the code simpler based on a few simple facts:

- The game is symmetric: what's good for one player is equally bad for the other one.
- (min x y) is the same as (- (max (- x) (- y)))

Together these facts have allowed him to write a simpler symmetric version of the minimax algorithm for tic-tac-toe. In this version, minimax always tries to maximize the current player's score. It doesn't have to pass an original player to a separate score function, and it doesn't have to switch back and forth between min and max functions.

Here's what Alan has:

```
(define (minimax bd cp)
  (cond [(has-three-in-a-row? bd) -1]           ; cp has lost
        [(has-no-empty-spots? bd)  0]           ; draw game
        [else
         (local [(define (loop alpha moves)     ; alpha is max score SO FAR
                   (cond [(empty? moves) alpha] ; for the next moves from bd
                         [else
                          (loop (max alpha
                                     (- (minimax (mark-x/o bd (first moves) cp)
                                                 (next-player cp))))
                                (rest moves))]))]
           (loop -2 (next-moves bd)))]))        ; alpha starts at -2, which is
                                                ; less than any real score
```

Alonzo comes by for a visit and Alan proudly shows him what he's done.

Alonzo says that it is nice, but points out two things:

- First, Alan's alpha parameter to loop is an accumulator. It is storing the max score so far as the loop proceeds through the possible moves from the current board.
- And second, there is a way to make minimax run much faster using TWO accumulators, alpha and beta, that are passed to both minimax itself and the loop, instead of just the loop. Alpha and beta represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. If beta becomes less than alpha the current branch of the search is aborted, since it can't result from best play by both players.

Alonzo also observes that there will probably be some inverting of the accumulators as happens with the args to max in Alan's code.

At this point (the ghost of) Ada comes along and says "Yes, that's the well-known (in Alonzo's time) alpha-beta search algorithm. Its amazing how easy it is to invent if you understand accumulators!"

Here's what you need to do:

Write out the code for alpha-beta, as a function with two accumulators that optimizes minimax as described above. It does not need any test cases, you can assume the existing tests for minimax will be used.