Rules Governing Formal Examinations

1. Each candidate must be prepared to produce, upon request, a UBCcard for identification.

2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions. No questions will be answered in this exam. If you see text you feel is ambiguous, make a reasonable assumption, write it down, and proceed to answer the question.

3. No candidate shall be permitted to enter the examination room after the expiration of one-half hour from the scheduled starting time, or to leave during the first half hour of the examination.

4. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action:
   - having at the place of writing any books, papers or memoranda, calculators, computers, sound or image players/recorders/transmitters (including telephones), or other memory aid devices, other than those authorized by the examiners;
   - speaking or communicating with other candidates; and
   - purposely exposing written papers to the view of other candidates or imaging devices. The plea of accident or forgetfulness shall not be received.

5. Candidates must not destroy or mutilate any examination material; must hand in all examination papers; and must not take any examination material from the examination room without permission of the invigilator.

6. Candidates must follow any additional examination rules or directions communicated by the instructor or invigilator.

Important notes about this examination

1. You have 90 minutes to write this examination.

2. Except for question 1, this exam will be graded significantly on how well you follow the design recipes. You have been given a copy of the Recipe Exam Sheet. Use it!

3. Put away books, papers, laptops, cell phones... everything but pens, pencils, erasers and this exam.

4. Good luck!
Problem 1

Given the following function definition:

```scheme
(define (foo x)
  (local [(define (bar i)
            (* x (foo i))
          (define (foo m)
            (* m x))]
  (bar x)))
```

Consider the evaluation of the following expression:

```scheme
(foo 2)
```

During this evaluation, two function definitions will be lifted. Write out just the lifted function definitions. (If it helps you to write out a more complete hand-stepping then feel free to do so, but be sure to clearly indicate the lifted function definitions.)
Problem 2

(A) Consider the following types comments, interpretations and data examples. Neatly annotate the type comments by doing 3 things:

- Draw an arrow from each type reference to the the corresponding type definition. All your arrows should have the ‘pointy-end’ at one of the names that appear right before ‘is’.
- Neatly label each arrow with one of MR, SR or R, depending on whether the reference is a mutual reference, self reference or an ordinary reference.
- Number each arrow 1, 2, 3 etc.; you will use these numbers in part B.

;; Description is String
;; Interp. the description of a parcel of land

(define-struct parcel (key desc next))
;; Parcel is (make-parcel Integer Description ListOfParcel)
;; Interp. a land parcel, with id, description and sub-parcels

;; ListOfParcel is one of
;;  - empty
;;  - (cons Parcel ListOfParcel)
;; Interp. an arbitrary number of Parcels
(define P1 (make-parcel 3 "cozy" empty))
(define P2
  (make-parcel 8 "wooded"
    (list (make-parcel 12 "ocean view" empty)
    (make-parcel 32 "water access"
    (list (make-parcel 90 "hilltop" empty))))))
(B) The template functions for the type comments in part A are below. For each call to another template function, label the call with the number of the corresponding arrow in part A.

(define (fn-for-desc d)
  (... d))

(define (fn-for-parcel p)
  (... (parcel-key p)
       (fn-for-desc (parcel-desc p))
       (fn-for-lop (parcel-next p))))

(define (fn-for-lop lop)
  (cond [(empty? lop) (...)]
       [else
        (... (fn-for-parcel (first lop))
             (fn-for-lop (rest lop)))]))
(C) A partial design for a function to add up all the keys in a parcel tree is shown below. You need to complete the design with tests and correct function definitions. We have given you the signature, purpose and a renamed copy of the template functions encapsulated with `local`. Fill in appropriate `check-expects` in the space provided, you may abbreviate `check-expect` as `c-e`. Then edit the renamed templates appropriately. Make your edits to the templates neatly, crossing out what you need to, and showing clearly what you would replace the ... with. Be sure to cross out anything you would not want in the final design.

;;; Parcel -> Natural

;;; Sum all the keys in a parcel tree

<put your check-expects here>

```scheme
(define (sum-keys p)
  (local [(define (sum-keys-desc d)
          (... d))
          (define (sum-keys-parcel p)
            (... (parcel-key p)
                (sum-keys-desc (parcel-desc p))
                (sum-keys-lop (parcel-next p))))
          (define (sum-keys-lop lop)
            (cond [(empty? lop) (...)]
                  [else
                   (... (sum-keys-parcel (first lop))
                        (sum-keys-lop (rest lop)))]))
          (sum-keys-parcel p)))
)```
Problem 3

The following functions are extracted from a world program that animates a fireworks show. Pick one of these functions, and show how to re-write it using one of the built in abstract list functions listed in the appendix to the exam. It is sufficient to just write the new body of the function. If you make marks on more than one of these functions circle your answer so we know which one to grade.

```lisp
(define (tick-fws lofw)
  (cond [(empty? lofw) empty]
        [else
         (cons (tick-fw (first lofw))
                (tick-fws (rest lofw)))]))

(define (remove-dead-fws lofw)
  (cond [(empty? lofw) empty]
        [else
         (if (alive? (first lofw))
             (cons (first lofw)
                   (remove-dead-fws (rest lofw)))
             (remove-dead-fws (rest lofw)))]))

(define (render-fws lofw)
  (cond [(empty? lofw) BG]
        [else
         (place-fw (first lofw)
                    (render-fws (rest lofw)))]))
```
Problem 4

Design a function called add-all that consumes a number \( n \), and a list of numbers \( \text{lon} \), and produces a list consisting of each element of \( \text{lon} \) with \( n \) added to it.

\[(\text{add-all} \ 3 \ (\text{list} \ 2 \ 1 \ 6)) \ \text{should produce} \ (\text{list} \ 5 \ 4 \ 9)\]

Your design must use one of the built-in abstract list functions (see appendix to exam). No marks will be awarded to a solution that uses the “old fashioned” template for a recursive function operating on a list.
Problem 5

Ron is writing functions that operate on binary trees. So far he has `mult-keys` and `append-vals`, and is about to write some more. Hermione comes in and says “You’re still not a wizard - you should design an abstract function from the two functions you have first! The function you design will be like a fold for BTs.” Ron isn’t sure what Hermione means about fold but he knows that he can extract an abstract function from two examples simply by following the process for doing so. He will figure out what the fold part means later.

```
(define-struct node (key val left right))
;; BT is one of:
;; - false
;; - (make-node Natural String BT BT)
;; a binary tree, each node has a key and value, and two children
(define BT1 false)
(define BT2 (make-node 1 "a"
    (make-node 6 "f"
        (make-node 4 "d" false false)
        false)
    (make-node 7 "g" false false)))

;; BT -> Natural
;; multiply the keys in a bt; produce 1 if bt is false
(check-expect (mult-keys BT1) 1)
(check-expect (mult-keys BT2) (* 1 6 4 7))

(define (mult-keys bt)
    (cond [(false? bt) 1]
        [else
            (* (node-key bt)
                (mult-keys (node-left bt))
                (mult-keys (node-right bt)))]))

;; BT -> String
;; produce a list of the values in a bt
(check-expect (append-vals BT1) ""
(check-expect (append-vals BT2) "afdg")

(define (append-vals bt)
    (cond [(false? bt) ""
        [else
            (string-append (node-val bt)
                (append-vals (node-left bt))
                (append-vals (node-right bt)))]))
```
(A) Write just the function definition (not the complete design) for an abstract function based on `mult-keys` and `append-vals` here. Call your function `fbt`.

(B) Write the “one-liner” function definition of `mult-keys` using `fbt` here.

(C) Ron is so happy with his new abstract function he shows it to Hermione and says he is going to use it to design `contains-key?` which consumes a key and a BT and produces true if the tree contains the key. Hermione says “No! `fbt` is not the right way to implement a function like `contains-key?`” Explain why `fbt` is not appropriate for this use. We are looking for a short and specific answer.
Problem 6

Design the function `merge`. It consumes two lists of strings, which it can assume are both sorted in ascending order using `string<=`. It produces a single list of all the strings, also sorted in ascending order. It should preserve duplicates, meaning that if a string occurs more than once in the lists it should also occur more than once in the result. Remember that both lists are already sorted, so your function does not need to re-sort the strings, it just needs to merge them properly.

Be sure to follow the recipe. Treat this as a function operating on two complex data. Your solution should explicitly show a cross product of type comments table, filled in with the result for each case. You do not need to include a stub, but you must include the other elements of a complete function design. As usual, you may abbreviate check-expect as c-e. Note that with respect to your function definition:

- A four case cond without simplified questions will receive limited credit.
- A four case cond with simplified questions will receive more credit.
- A three case cond with simplified questions will receive even more credit.
Continue problem 6 here.
Use this page for extra space, clearly label any problem solution you write here.
Use this page for extra space, clearly label any problem solution you write here.
Use this page for extra space, clearly label any problem solution you write here.