CS107 Final Exam Practice Problem Solutions

Solution 1: Matchmaking

```
/**
 * The primary assumption is that both boys and girls
 * are C vectors of dynamically allocated C strings, each
 * initialized as follows.
 *
 *   vector boys, girls;
 *   VectorNew(&boys, sizeof(char *), StringFree, 0);
 *   VectorNew(&girls, sizeof(char *), StringFree, 0);
 *
 * generateAllCouples creates a new C vector of couples
 * and inserts one such record on behalf of every possible
 * mapping of boy to girl.  The couples own their own strings,
 * so that none of the three vectors share any memory whatsoever.
 * Assume that CoupleFree is the VectorFreeFunction that disposes
 * of couple records embedded in a vector, and assume it just works.
 */

typedef struct {
   char *girl;
   char *boy;
} couple;

vector generateAllCouples(vector *boys, vector *girls)
{
   vector couples;
   VectorNew(&couples, sizeof(couple), CoupleFree, 0);
   int i, j;
   couple item;

   for (int i = 0; i < VectorLength(boys); i++) {
      for (int j = 0; j < VectorLength(girls); j++) {
         item.boy = strdup(*(char **) VectorNth(boys, i));
         item.girl = strdup(*(char **) VectorNth(girls, j));
         VectorAppend(&couples, &item);
      }
   }

   return couples;
}
```

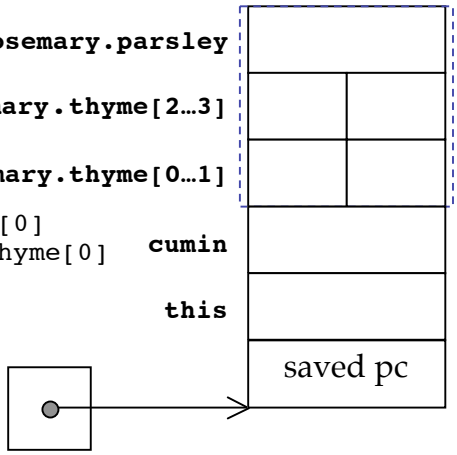
Solution 2: Extending the `vector`

```
typedef struct {
   void *elems;         // pointer to elemsize * alloclength bytes of memory
   int elemsize;        // number of bytes dedicated to each client element
   int loglength;       // number of elements the client is storing
   int alloclength;     // number of elements we have space for
   VectorFreeFunction free;      // applied to elements as they are removed
} vector;

typedef bool (*VectorSplitFunction)(const void *elemAddr);
void VectorSplit(vector *original, vector *thoseThatPass,
                 vector *thoseThatFail, VectorSplitFunction test)
```

```
{
   int i;
   void *elem;

   VectorNew(thoseThatPass, original->elemsize, original->free, 0);
   VectorNew(thoseThatFail, original->elemsize, original->free, 0);
   for (i = 0; i < original->loglength; i++) {
      elem = VectorNth(original, i);
      VectorAppend((test(elem) ? thoseThatPass : thoseThatFail), elem);
   }

   original->loglength = 0;  // leave memory there…
}
```

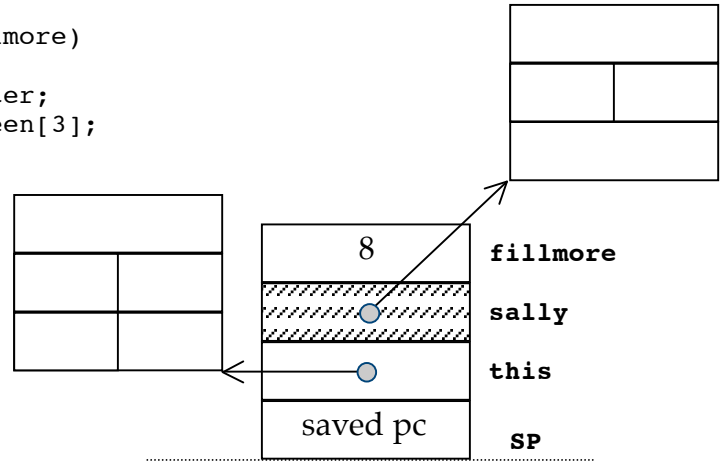
Solution 3: Spice Rack

```
class spice {

   spice *& saffron(spice& salt);
   short sage(int cumin, spice rosemary) {
line 1    cumin *= thyme[cumin - *(char *)thyme];
line 2    return ((spice *) &parsley)->saffron(rosemary)->parsley - &rosemary;
   }

   short thyme[4];
   spice *parsley;
};

// line 1
R1 = M[SP + 8];      // load old cumin
R2 = M[SP + 4];      // load this aka &this->thyme[0]
R3 =.1 M[R2];        // load first byte of this->thyme[0]
R4 = R1 – R3;        // compute index
R5 = R4 * 2;         // scale by sizeof(short)
R6 = R2 + R5;        // compute address of rhs
R7 =.2 M[R6];        // compute rhs
R8 = R1 * R7;        // compute new cumin value
M[SP + 8] = R8;      // flush to space for cumin

// line 2
R10 = M[SP + 4];     // load this (again)
R11 = R10 + 8;       // compute &this->parsley (and look, it's a spice * now)
R12 = SP + 12;       // prepare address of rosemary (which gets passed by ref)
SP = SP – 8;         // make space for params
M[SP] = R11;         // set down address of receiving object
M[SP + 4] = R12;     // set down address backing salt reference
CALL <spice::saffron>
SP = SP + 8;         // clean up params
R13 = M[RV];         // RV has spice ** backing spice *&, load real spice *
R14 = M[R13 + 8];    // load value of parsley
R15 = SP + 12;       // load &rosemary
R16 = R14 – R15;     // compute raw number of bytes in between the two
RV = R16 / 12;       // scale back to quantum number of spice records between them
RET;
```



Solution 4: Cars

```
   class car {

      char **operator[](const char *);
      car& dochudson(car& sally, int fillmore)
      {
line 1   sally["Pixar"][fillmore] += *mater;
line 2   return (*(car **)mcqueen)->mcqueen[3];
      }

      short mater[4];
      car *mcqueen;
   };


   // sally["Pixar"][fillmore] += *mater;
   R1 = M[SP + 8];
   SP = SP – 8;
   M[SP] = R1;
   M[SP + 4] = 1000;
   CALL <car::operator[]> // RV has a char **
   SP = SP + 8;

   R1 = M[SP + 12];  // load fillmore
   R2 = R1 * 4;      // scale fillmore by sizeof(char *)
   R3 = RV + R2;     // compute address of char * to be advanced
   R4 = M[SP + 4];   // load this
   R5 =.2 M[R4];     // load this->mater[0]
   R6 = M[R3];       // load old value of char *
   R7 = R6 + R5;     // compute new value of char *
   M[R3] = R7;       // done!

   // return (*(car **)mcqueen)->mcqueen[3];
   R1 = M[SP + 4];   // load this (yes, again!)
   R2 = M[R1 + 8];   // load this->mcqueen (and look.. it's a car **)
   R3 = M[R2];       // load *(car **)(this->mcqueen)
   R4 = M[R3 + 8];   // load *(car **)(this->mcqueen)->mcqueen
   RV = R4 + 36;     // populate RV *(car **)(this->mcqueen)->mcqueen + 3
   RET;              // leave ☺
```



Solution 5: Marriage And Mapping

a)

```
(define (marry singles-list)
  (if (<= (length singles-list) 1) singles-list
      (cons (list (car singles-list) (cadr singles-list))
            (marry (cddr singles-list)))))
```

b)

```
(define (map-everywhere func structure)
  (map (lambda (elem)
         (if (list? elem) (map-everywhere func elem)
             (func elem)))
       structure))
```

Solution 6: Longest Common Subsequences

a)

```
;;
;; Function: longest-common-prefix
;; -------------------------------
;; Takes two lists and returns the longest prefix common
;; to both of them.  If there is no common prefix, then
;; longest-common-prefix evaluates to the empty list
;;
;; Examples:
;;  (longest-common-prefix '(a b c) '(a b d f)) --> (a b)
;;  (longest-common-prefix '(s t e r n) '(s t e r n u m)) --> (s t e r n)
;;  (longest-common-prefix '(1 2 3) '(0 1 2 3)) --> ()
;;

(define (longest-common-prefix seq1 seq2)
  (cond ((or (null? seq1) (null? seq2)) '())
        ((not (equal? (car seq1) (car seq2))) '())
        (else (cons (car seq1)
                    (longest-common-prefix (cdr seq1) (cdr seq2))))))
```

b)

```
;;
;; Function: mdp
;; -------------
;; Mapping routine which transforms a list of length n into another
;; list of length n, where each element of the new list is the result
;; of levying the specified func against the corresponding cdr of
;; the original.
;;
;; Examples:
;;  (mdp length '(w x y z)) --> (4 3 2 1)
;;  (mdp cdr '(2 1 2 8)) --> ((1 2 8) (2 8) (8) ())
;;  (mdp reverse '("ba" "de" "foo" "ga")) -->
;;           (("ga" "foo" "de" "ba") ("ga" "foo" "de") ("ga" "foo") ("ga"))
;;

(define (mdp func sequence)
  (if (null? sequence) '()
      (cons (func sequence) (mdp func (cdr sequence)))))
```

c)

```
;;
;; Function: longest-common-sublist
;; --------------------------------
;; Analyzes the two sequences and computes the longest sublist that's
;; common to both of them.  If there are no common elements at all, then
;; the empty list is returned.

(define (longest-common-sublist seq1 seq2)
  (car (quicksort (generate-all-sublists seq1 seq2) list-length>?)))

;;
;; Function: generate-all-sublists
;; -------------------------------
;; Uses double mdpping to pair every suffix of the
;; first sequence with every suffix of the second,
```

```
;; generating the longest prefix common to each of them.
;; The apply append is needed, because each cdr is mapped
;; to a list of all sublists that are prefixes of that
;; cdr.
;;

(define (generate-all-sublists seq1 seq2)
  (apply append (mdp (lambda (suffix1)
                       (mdp (lambda (suffix2)
                              (longest-common-prefix suffix1 suffix2))
                            seq2))
                     seq1)))

;;
;; Function: list-length>?
;; -----------------------
;; Returns #t if and only if the first list
;; has more top-level elements than the second,
;; and returns #f otherwise.
;;

(define (list-length>? ls1 ls2)
  (> (length ls1) (length ls2)))
```

Solution 7: File Sharing

```
int DownloadMediaFile(const char *server, const char *file);

int DownloadMediaLibrary(const char *server, const char *files[], int numFiles)
{
   int i, totalNumBytes = 0;
   Semaphore byteCountLock = SemaphoreNew("Byte Count Lock", 1);
   Semaphore numThreadsAllowed = SemaphoreNew("Threads Allowed", 12);
   Semaphore numThreadsCompleted = SemaphoreNew("Num Threads Completed", 0);

   for (i = 0; i < numFiles; i++)
      ThreadNew("Downloader, DownloadThread, 6, server, files[i],
                             &totalNumBytes, byteCountLock,
                             numThreadsAllowed, numThreadsCompleted);

   for (i = 0; i < numFiles; i++)
      SemaphoreWait(numThreadsCompleted);

   return totalNumBytes;
}

void DownloadThread(const char *server, const char *filename,
                    int *totalByteCountp, Semaphore byteCountLock,
                    Semaphore numThreadsAllowed, Semaphore numThreadsCompleted)
{
   int numBytes;

   SemaphoreWait(numThreadsAllowed)
   numBytes = DownloadMediaFile(server, filename);
   SemaphoreSignal(numThreadsAllowed);

   SemaphoreWait(byteCountLock);
   *totalByteCountp += numBytes;
   SemaphoreSignal(byteCountLock);
   SemaphoreSignal(numThreadsCompleted);
}
```

Solution 8: Concurrent, Short-Circuit Evaluation of Scheme's and

```
typedef struct {
   enum { Boolean, Integer, String, Symbol, Empty, List} type;
   char value[8];  // value[0] stores '\0' for #f, anything else for #t
   // above eight bytes are general-purpose bytes…
} Expression;

typedef struct {
   Semaphore lock;
   Semaphore answerReady;
   Semaphore answerAccepted;
   Expression *answer;
   int numChildrenRemaining;
} AndExpressionInfo;

/**
 * Function: evaluateConcurrentAnd
 * -------------------------------
 * Special function dedicated to the implementation of the
 * concurrent-and special form.  It returns the first #f Expression
 * ever produced by a child, or if #f is never produced, then it
 * returns the last Expression * produced by the last thread
 * to complete.
 *
 * @param exprs an array of Expressions * to be concurrently evaluated.
 *              We assume there are no recursive calls to concurrent-and
 *              involved.
 * @param n the length of the exprs array
 * @return the result of the last child thread needed in order to produce
 *         an answer.
 */

Expression *evaluateConcurrentAnd(Expression *exprs[], int n)
{
   AndExpressionInfo *info = malloc(sizeof(AndExpressionInfo));
   info->lock = SemaphoreNew("Lock", 1);
   info->answerReady = SemaphoreNew("Answer Available", 0);
   info->answerAccepted = SemaphoreNew("Answer Accepted", 0);
   info->answer = NULL;
   info->numChildrenRemaining = n;

   for (int i = 0; i < n; i++) {
      char threadName[128];
      sprintf(threadName, "Sub-expression Thread %d", i + 1);
      ThreadNew(threadName, evaluateExpressionThread, 2, exprs[i], info);
   }

   SemaphoreWait(info->answerReady);
   Expression *answer = info->answer;
   SemaphoreSignal(info->answerAccepted);
   return answer;
}

void evaluateExpressionThread(Expression *expr, AndExpressionInfo *info)
{
   Expression *result = evaluateExpression(expr);
```

```
    SemaphoreWait(info->lock);
    bool lastToFinish = (--info->numChildrenRemaining == 0);
    if (info->answer == NULL) {
       if (lastToFinish ||
           result->type != Expression::Boolean || result->value[0] == '\0') {
          info->answer = result;
          SemaphoreSignal(info->answerReady);   // signal parent
          SemaphoreWait(info->answerAccepted);  // stall until parents reads answer
       }
    }
    SemaphoreSignal(info->lock);

    if (lastToFinish) { // must free here (or at least outside the parent thread)
       SemaphoreFree(info->lock);
       SemaphoreFree(info->answerReady);
       SemaphoreFree(info->answerAccepted);
       free(info);
    }
}
```