CS107 Final Exam Practice Problems

Exam Facts

- First Offering: Monday, June 9th at 8:30 a.m. in Dinkelspiel Auditorium.
- Second Offering: : Monday, June 9th at 3:30 p.m. in Hewlett 200.
- Three hours, open notes, open book, closed computer.

You may take the exam at whichever is the more convenient of these two times. I'll provide documentation for any relevant C, C++, and Scheme built-ins. The final exam I'm presenting here contains many more problems than you'll see on your final exam. I thought it would be better to give you plenty of practice problems—all drawn from previous midterms and finals—that'll help you identify the material I consider important.

SCPD students will take the exam much like they took the midterm. The exam will be posted as a handout at 3:30 p.m. I need the exam to be faxed in by Tuesday at 5:00 p.m. My cell phone and fax number will be posted on the first page of the exam, so you know where to call if you have questions and where to fax when you're all done.

Material

The final is comprehensive but will emphasize topics covered after the midterm. Want to see where you've been? Here's the pretty impressive list of things you've learned in 107:

1. Implementation—stack-heap diagrams, memory layout, structures, arrays and pointers, function calls, parameter passing, local variables, C and C++ code generation.
2. C— arrays, pointers, **`malloc`**, **`&`**, **`*`**, **`void*`**, typecasts, function pointers, preprocessor, compiler, linker.
3. Simple Concurrency—threads, semaphores, binary lock, rendezvous, shared global data, race conditions.
4. Scheme—lists, **`car`**, **`cdr`**, **`cons`**, **`append`**, **`map`**, **`lambda`**, **`apply`**, **`let`**.

Remember: No Python the exam.

Problem 1: Matchmaking

You keep the names of all your male friends in one C **vector** (where else?) and the name of all your female friends in a second C **vector**. Your task is to generate a brand new **vector** and populate it will the full cross product of men and women as **char ***, **char *** pairs.

Write a function **generateAllCouples** that creates a **vector**, inserts deep copies of all boy-girl pairs, and then returns it. Note that this problem has no C++ component whatsoever. It's all C.)

```
/**
 * 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);
```

Problem 2: Extending the vector

A properly implemented C **vector** should have completed the **struct** as follows:

```
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;
```

Pretend that we've decided to extend the **vector** abstraction to include one more function: **VectorSplit**. **VectorSplit** initializes and populates two additional **vector**s, copying those elements that pass a supplied predicate function to the first **vector**, and copying those that fail to the second. The original **vector** is completely depleted of its elements without actually destroying them.

Your implementation has access to the **vector** fields, and can make use of the other **vector** functions if you so choose. In essence, you're emptying out the original **vector** and partitioning its elements so that some end up in the first, and the rest end up in the second. Be careful to not free anything you're not supposed to. This is pure C, so no C++ is allowed.

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

Problem 3: C++ Spice Rack

Given the following C++ **class** definition, generate code for the **spice::sage** method. Assume that the parameters have already been set up for you. Be clear about what code pertains to which line. Recall that C++ references are automatically dereferenced pointers, and k-argument methods are really (k + 1)-argument functions, where the address of the receiving object is quietly passed in as the bottommost parameter. The address of the first instruction of the saffron method is synonymous with <spice::saffron>. You have this and the next page for your code.

```
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;
};
```

Problem 4: Cars

Given the following C++ **class** definition, generate code for the **car::dochudson** method. Assume that the parameters have already been set up for you, and don't worry about returning from the method. Be clear about which code pertains to which line. Recall that C++ references are automatically dereferenced pointers, and k-argument methods are really (k + 1)-argument functions, because the address of the receiving object is quietly passed in as the bottommost parameter. The address of the first instruction of the **operator[]** method is synonymous with **<car::operator[]>**. Assume that the **'P'** of **"Pixar"** is at address constant **1000**.

```
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;
};
```

Problem 5: Marriage And Mapping

a) Write a Scheme function called **marry**, which takes a list of strings and bundles neighboring pairs into sublists.

```
> (marry '("Adam" "Eve" "George" "Martha" "Albert" "Costello"))
(("Adam" "Eve") ("George" "Martha") ("Albert" "Costello"))
> (marry '("Fred" "Wilma"))
(("Fred" "Wilma"))
> (marry '())
()
```

If there are an odd number of strings, then the last string is left as is, because he's a total loser.

```
> (marry '("Lucy" "Schroeder" "Patty" "Linus" "Charlie Brown"))
(("Lucy" "Schroeder") ("Patty" "Linus") "Charlie Brown")
> (marry '("INFP"))
("INFP")

(define (marry singles)
   (
```

b) Now write a function called **map-everywhere**, which traverses a heterogeneous list (of integers, strings, doubles, and/or nested heterogeneous lists), and evaluates to the same exact structure, where every **atom** has been transformed by the specified function. You may assume that there are no empty lists anywhere, and that the initial argument to **map-everywhere** is always a list.

```
> (map-everywhere (lambda (x) (+ x 1)) '(1 (2 (3) 4 (5 (6))) (8 9)))
(2 (3 (4) 5 (6 (7))) (9 10))
> (map-everywhere list '("a" ("b" ("c" ("d" "e" "f" "g")))))
((a) ((b) ((c) ((d) (e) (f) (g)))))
> (map-everywhere factorial '(4 ((5)) (6 (7))))
(24 ((120)) (720 (5040)))

(define (map-everywhere func structure)
   (
```

Problem 6: Longest Common Subsequences

a) Write a routine called **longest-common-prefix**, which takes two arbitrary lists and computes the longest prefix common to both of them. You should assume that each of the two arguments is a list, and that the front two elements must be equivalent in the **equal?** sense in order for them to contribute to a common prefix.

```
;;
;; 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)
   (
```

b) Write a routine called **mdp**, which is like (the unary version of) **map**, except that the specified **func** is applied to the series of non-null **cdr**s of the specified list. The products of the calls to **func** are bundled in a list.

```
;;
;; 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)
   (
```

c) Using **longest-common-prefix**, **mdp**, and **quicksort**, write a routine called **longest-common-sublist**, which takes two non-empty lists and returns the longest sublist common to both of them. If there are two or more sublists of maximal length, then any one of them may be returned. You should not use any exposed **car**-**cdr** recursion, but instead rely on the **mdp** to match all **cdr**s of the first sequence against all **cdr**s of the second.

```
;;
;; 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.
;;
;; Examples:
;;  (longest-common-sublist '(a b d f g j) '(b c d f g h j k)) --> (d f g)
;;  (longest-common-sublist '(a b c d) '(x y)) --> ()
;;  (longest-common-sublist '(4 5 6 7 8 9 10) '(2 3 4 5 6 7 8)) --> (4 5 6 7 8)
;;
;; Assume the version of quicksort you wrote for Assignment 8 just works:
;;
;;    (define (quicksort sequence comp)
;;       (...

(define (longest-common-sublist seq1 seq2)
   (
```

Problem 7: File Sharing

Millions of people around the globe download audio and video files on a daily basis. Web sites like **napster**, **youtube**, **myspace**, and **itunes** all offer a variety of digital media products. You've decided to put your programming skills to work and write a function to concurrently download your full wish list of songs and movies.

The following routine is thread-safe and has already been written for you. It downloads the contents of the specified file from the specified server and returns the total number of bytes downloaded.

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

Your job is to implement the **DownloadMediaLibrary** function, which takes a server name and an array of file names, downloads all files using the **DownloadMediaFile** routine, and returns the total number of bytes downloaded. For each file in the array, you should spawn off a separate thread to download its contents. However, because you don't want to pelt the file server with an unreasonable number of simultaneous requests, you should limit the number of active connections to 12, so that the number of concurrently executing calls to **DownloadMediaFile** will never be more than 12 at any single instant. You must ensure that all calls to **DownloadMediaFile** have completed before **DownloadMediaLibrary** returns.

You should assume that **InitThreadPackage** and **RunAllThreads** have already been called. Declare shared integers and Semaphores locally, being clear what initial value should be attached to them.

```
int DownloadMediaLibrary(const char *server, const char *files[], int numFiles)
{
```

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

Symbols like **+**, **append**, **car**, and **cons** are bound to standard procedures, because those procedures require that all of their arguments be evaluated in order to synthesize a result. Special forms like **if**, **cond**, **and**, and **or** are different in that they often require only a subset of their arguments to be evaluated in order to produce an answer. Depending on the outcome of a test, **if** evaluates one expression and ignores a second one. **cond** evaluates only as many tests as are needed to produce something other than **#f**. **and** and **or** short-circuit the instant they can generate **#f** and **#t**, respectively.

Using the **Expression** definition given here

```
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;
```

we're going to assume that a sequential, thread-safe function called **evaluateExpression** is provided to work for all expression evaluations. All means it works when fed a primitive like **4**, a quoted list like **'(scheme is clever)**, a standard procedure call like **(cons 1 '(2 3))**, or an expression with a special form at the front like **(if (< x y) x y)**.

```
Expression *evaluateExpression(Expression *expr);
```

Consider the introduction of a **new** special form called **concurrent-and**. Like **and**, it returns **#f** if and only if one of its arguments evaluates to **#f**. Like the built-in **and**, it returns **#f** the instant it detects that one of its expression arguments evaluated to **#f**.

But **concurrent-and** is different: it evaluates each of its arguments simultaneously—or seemingly so—by relying on the thread package we used for Assignment 6. Because **concurrent-and** involves threading, **evaluateExpression** all by itself isn't what we want. We want to implement a routine called:

```
Expression *evaluateConcurrentAnd(Expression *exprs[], int n);
```

evaluateConcurrentAnd is the C function that implements the **concurrent-and** special form. It spawns **n** threads, one for each **Expression ***, and each thread makes use of **evaluateExpression** to generate a result. **evaluateConcurrentAnd** then waits while the threads do their work. As each child thread finishes, it communicates its result back to **evaluateConcurrentAnd**. If **#f** comes back, then **evaluateConcurrentAnd** immediately returns that **#f** and lets all active child threads go on to compute results that don't matter any more. If **evaluateConcurrentAnd** produces something other than **#f** (like **#t**, or an expression on any non-Boolean type) then it did so because it waited for all **n** child threads to complete, and not a single one of them saw its sub-expression evaluate to **#f**.

Notice that the short-circuit evaluation is different here. Like the built-in **and**, **concurrent-and** returns as soon as it can. Unlike the built-in, it evaluates all of the sub-expressions, even if it doesn't wait to hear what every single result turned out to be. It's still short-circuit evaluation—it's just a different flavor.

You should assume that **evaluateExpression** is thread-safe and that **RunAllThreads** has already been called. You should implement **evaluateConcurrentAnd** according to the above description. You should not use any global variables whatsoever, and any dynamically allocated memory should be freed before returning (or eventually freed by the threads that don't matter any more.) No deadlock. No race conditions. All **Semaphore**s should initialized using **SemaphoreNew** and should be disposed of using **SemaphoreFree**.

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

/**
 * 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)
{
```