Exam Facts

- First Offering: Friday, June 8th at 8:30 a.m. in Cubberly Auditorium.
- Second Offering: Friday, June 8th at 7:00 p.m. in Cubberly Auditorium.
- 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++, Scheme, and Java built-ins, so don’t go memorizing anything. 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. I repeat: This is much longer than anything you’ll see on Friday June 8th.

SCP D students are encouraged to come to campus if at all possible. Those taking it remotely can take it on Friday evening at 7:00 p.m. (when it’ll be posted as a handout) or on Saturday morning. I’ll have my cell phone one me (number listed on exam) and you’ll fax the exam in (fax number also on the exam). Just make sure I have your exam by noon on Saturday, because we’ll need to grade it by then. 😊

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, 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.
5. Java—inheritance, constructors, static methods, overriding, super, code factoring, protected.
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 vectors, 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.

```c
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.

```c
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[]>.

```c
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 \texttt{marry}, which takes a list of strings and bundles neighboring pairs into sublists.

\[
\texttt{> (marry '("Adam" "Eve" "George" "Martha" "Albert" "Costello"))}
\]
\[
\texttt{("Adam" "Eve") ("George" "Martha") ("Albert" "Costello")}
\]
\[
\texttt{> (marry '("Fred" "Wilma"))}
\]
\[
\texttt{("Fred" "Wilma")}
\]
\[
\texttt{> (marry '()))}
\]

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

\[
\texttt{> (marry '("Lucy" "Schroeder" "Patty" "Linus" "Charlie Brown"))}
\]
\[
\texttt{("Lucy" "Schroeder") ("Patty" "Linus") "Charlie Brown"}
\]
\[
\texttt{> (marry '("INFP"))}
\]
\[
\texttt{("INFP")}
\]

\[
\texttt{(define (marry singles)}
\]
\[
\texttt{)}
\]

b) Now write a function called \texttt{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 \texttt{map-everywhere} is always a list.

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

\[
\texttt{(define (map-everywhere func structure)}
\]
\[
\texttt{)}
\]

Problem 6: Longest Common Subsequences

a) Write a routine called \texttt{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 \texttt{equal?} sense in order for them to contribute to a common prefix.

\[
\texttt{;;}
\]
\[
\texttt{;; Function: longest-common-prefix}
\]
\[
\texttt{;; -------------------------------}
\]
\[
\texttt{;; Takes two lists and returns the longest prefix common}
\]
\[
\texttt{;; to both of them. If there is no common prefix, then}
\]
\[
\texttt{;; longest-common-prefix evaluates to the empty list}
\]
\[
\texttt{;;}
\]
\[
\texttt{;; Examples:}
\]
\[
\texttt{;; (longest-common-prefix '(a b c) '(a b d f)) --> (a b)}
\]
\[
\texttt{;; (longest-common-prefix '(s t e r n) '(s t e r n u m)) --> (s t e r n)}
\]
\[
\texttt{;; (longest-common-prefix '(1 2 3) '(0 1 2 3)) --> ()}
\]
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 `cdrs` of the specified list. The products of the calls to `func` are bundled in a list.

```
(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 `cdrs` of the first sequence against all `cdrs` of the second.

```
(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, 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 Semaphores 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.
Problem 9: Getting the Housing Draw Right!

One of the great traumas of spring quarter is planning for next year's housing—deciding whether to live on or off campus and competing in the lottery for the precious on-campus spaces.

On the next page, you'll find code for the starting implementation of a generic **Student** class. A student responds to messages to take a chance in the housing draw, handle living on campus or off, and tracks a few facts: whether the student is male or female, if this year is guaranteed, the student’s bank balance and their morale. The morale is expressed as an integer (higher is better, 0 or negative is bad). Living on campus, for example, improves a student’s morale.

We’re interested in what happens when you send an **arrangeForHousing** message to a Student object. A student decides whether to bypass the draw entirely and choose to live off-campus or enter the lottery for on-campus housing. If a student doesn’t get assigned, they live off-campus, but not by choice. After determining out the result, **arrangeForHousing** calls the appropriate live on or off campus method for the student.

Students have various probabilities of getting assigned in the on-campus draw. All students track their "guaranteed" status. If guaranteed, a student has a good chance of being assigned. For example, a guaranteed Grad student has a 90% chance of getting on-campus in the draw, but only 50% if unguaranteed. A guaranteed Undergrad has a 95% chance, while unguaranteed is 30%. Undergrads also track an additional "preferred" status that is even better than guaranteed—if an Undergrad is preferred, they have a 98% chance. All of these probabilities are for male students. Without regard to type or status, a female student always has a 2% better chance of getting assigned than her male equivalent.

The students differ in some other ways. For example, Undergrads really enjoy living on campus and thus always choose to enter the draw and if they end up on campus, their morale improves by an extra 10 points. Undergrads who end up living off-campus hold lots of parties to improve their lonely social life, causing their bank balance to go down by $500.

Graduate students will choose to enter the draw if the cost of on-campus housing is less than the off-campus cost, otherwise they happily move off campus. The Graduates also keep one shared list of all frustrated Graduate students who unwillingly had to move off campus. Whenever a multiple of 100 students is reached on this list, all students on the list are sent a "protest" message. When a Graduate student protests, they chant "More housing!" three times (that adds up to a lot of chanting for all frustrated students). In addition, if the Graduate student has some cash lying around (a bank balance of more than $500), they spend $100 buying materials to make protest posters. Protesting improves a Grad student’s morale by 5 points.
In most ways, Masters candidates behave like other Graduate students. However in deciding whether to enter the draw, they also take into consideration their bank balance. If their bank balance is negative, they enter the draw, figuring it's easier to owe the university money than a landlord. When a Masters student is asked to protest, they only join in with the chanting and poster-making if their own morale was negative, otherwise they just skip it.

Your job is to design and implement the student classes as described above. You are free to add any other helper classes and can change or add to the generic Student class as well.

Note that we are not going to worry about allocation or initialization. You do not have to write any constructors. Where needed you can indicate the starting value for variables. Your most important design goal is to avoid code duplication. It is recommended you think through the entire design before making any decisions.

```java
public class Student {
    // This helper is used to determine if a student gets assigned in the housing draw. If prob is .75, returns true 75% of time, false 25%.
    protected boolean gotAssignedInDraw(double probability)
    {
        return (probability >= Math.random());
    }

    // instance and class variables, assume they are set with correct values as needed
    protected int morale, bankBalance;
    protected boolean isMale, isGuaranteed;
    protected static int OnCampusRent, OffCampusRent;
}
```

Make it clear what modifications you want to make to the above Student class. You can change the code inside the given methods, add new methods, add new variables, make things abstract, change the access specifiers, etc. whatever you need to get the job done.
Problem 10: Java Runtime

You are given the following class definitions:

```java
public abstract class Rock {
    public abstract void georgia();
    public void lou() { echo("Rock.lou"); georgia(); }
    public void mary() { echo("Rock.mary"); murray(); }
    public static void murray() { echo("Rock.murray"); }
    public void ted() { echo("Rock.ted"); }
    protected static void echo(String text) { System.out.println(text); }
}

public class Igneous extends Rock {
    public void georgia() { echo("Igneous.georgia"); mary(); }
    public static void murray() { echo("Igneous.murray"); }
}

public abstract class Sedimentary extends Rock {
    public void mary() { echo("Sedimentary.mary"); ted(); }
    public static void murray() { echo("Sedimentary.murray"); }
    public void ted() { echo("Sedimentary.ted"); murray(); }
}

public class Metamorphic extends Sedimentary {
    public void georgia() { echo("Metamorphic.georgia"); mary(); }
    public void lou() { echo("Metamorphic.lou"); super.lou(); }
    public static void murray() { echo("Metamorphic.murray"); }
    public void ted() { echo("Metamorphic.ted"); super.ted(); }
}
```

Consider the following method that is declared to take a `Rock` object as a parameter:

```java
void sueannivens(Rock paperweight) {
    paperweight.murray();
    paperweight.lou();
}
```

What are the possible types of objects that `paperweight` may be referencing at runtime? For each possibility, trace through a call to the `sueannivens` method and show the output that would be printed.
Problem 11: OOP Design Problems

a. You're designing four classes (Apple, Banana, Cherry, and Date), all of which respond to the same three methods: seed, rind, and flesh.

- The functionality attached to the seed method is the same for all four classes.
- The functionality associated with the rind method is the same for all classes except Cherry, which has a completely unrelated implementation for rind.
- All four classes have independent implementations for the flesh method.

Design the four classes so that the maximum amount of code is unified. Express your design in the form of a tree diagram like those used in lecture and in the handouts. Be sure to mark classes and methods as abstract when appropriate, and be sure to repeat the method within a subclass when you intend to provide an implementation. You needn’t be worried with access modifiers such as public, private, or protected. You needn’t actually implement any methods, as we’re only interested in the design.

b. You are designing eight classes (Mary, David, Joanie, Nancy, Elizabeth, Susan, Tommy, Nicholas), and all respond to the same three non-static methods: tom, joan, and abby.

- Mary, David, Joanie, and Nancy all behave identically in response to the tom message, whereas Elizabeth, Susan, Tommy, and Nicholas also behave identically (though differently than Mary, David, Joanie, and Nancy) in response to the tom message.
- Mary, David, and Joanie all respond in exactly the same way when sent a joan message, but Nancy, Elizabeth, Susan, Tommy, and Nicholas all behave differently from the first three, and differently from each other.
- All classes respond to the same exact implementation of abby, save Mary and Nicholas, who each need an implementation specific to themselves (and independent from each other’s).

Design the eight classes so that the maximum amount of code is unified. Express your design in the form of a tree diagram like those used in lecture and in the handouts. Be sure to mark classes and methods as abstract when appropriate, and be sure to repeat the method within a subclass when you intend to provide an implementation. You needn’t be worried with access modifiers such as public, private, or protected. You needn’t actually implement any methods, as we’re only interested in the design.