Exam 2 will be a full–period exam on Thursday, 4 April. The exam will be taken entirely from the following questions (although some of the questions may be changed in minor ways).

1. All questions for Test 1 (from Handout 10).
2. All questions from the Functional Parameters handout (Handout 13).
3. All questions from the Stacks and Queues handout (Handout 16).
4. Answer the following True/False questions:

   ________ A mutator function is used for its side–effect.

   ________ In our approach to linked lists, the expression `cons(42, foo)` increases the length of the list contained in the variable `foo` by exactly one.

   ________ The expression `rest(rest(foo))` returns the second data element in the list `foo`.

   ________ The expression `rest(rest(foo))` returns the third data element in the list `foo`.

   ________ A stack is a recursive implementation of a linked list.

   ________ A stack adheres to a LIFO (last–in/first–out) discipline.

   ________ It is possible for the first item pushed onto a stack to be the first item subsequently popped off of that stack.

   ________ It is possible for the first item enqueued into a queue to be different from the first item subsequently dequeued from that queue.

   ________ A queue adheres to a FIFO (first–in/first–out) discipline.

5. Briefly (*i.e.*, in 1–3 sentences) define each of the following terms:
   - Data abstraction
   - Predicate
   - Mutator
   - Functional parameter
   - Side–effect

6. Assume that each of the following pieces of code is embedded within a legal C program. Show what is output by the `printf` statement(s) in each code fragment. Assume that stacks and queues have been defined as we defined them in class, that `stack_item_type` and `queue_item_type` are both `int`, that `stack` and `queue` are variables of type `stack_type` and `queue_type` respectively, and that `i` is an integer variable. Recall that the operations defined on stacks are `create_stack`, `push`, `pop`, and `empty_stack` and that the operations defined on queues are `create_queue`, `enqueue`, `dequeue`, and `empty_queue`.

-

```
stack = create_stack();
queue = create_queue();
for (i = 2; i < 6; i++)
  enqueue(i, queue);
while (!empty_queue(queue)) {
  enqueue(dequeue(queue), queue);
  push(dequeue(queue) + 1, stack);
}
while (!empty_stack(stack))
   printf("%d ", pop(stack));
```

-

```
stack = create_stack();
queue = create_queue();
for (i = 3; i < 8; i++) {
  if (i % 2 == 0)
    push(i, stack);
  else enqueue(i, queue);
}
while (!empty_stack(stack))
  enqueue(pop(stack) - dequeue(queue), queue);
while (!empty_queue(queue))
   printf("%d ", dequeue(queue));
```

-

```
stack = create_stack();
queue = create_queue();
for (i = 1; i <= 4; i++)
   push(i, stack);
for (i = 5; i < 8; i++)
   enqueue(i - pop(stack), queue);
while (!empty_queue(queue)) {
   push(dequeue(queue) + 1, stack);
   if (!empty_queue(queue))
      enqueue(dequeue(queue) + 1, queue);
}
while (!empty_stack(stack))
   printf("%d ", pop(stack));
```

-

```
stack = create_stack();
queue = create_queue();
for (i = 7; i > 0; i = i - 2) {
   push(i + 1, stack);
   enqueue(i, queue);
   enqueue(dequeue(queue), queue);
}
while (!empty_stack(stack))
   enqueue(pop(stack) + dequeue(queue), queue);
while (!empty_queue(queue))
   printf("%d ", dequeue(queue));
```

For the following questions, assume that lists, stacks and queues have been defined as we defined them in class. For lists, that means that the types `list_type` and `list_item_type`, the functions `first`, `rest`, `cons` and `empty_list`, and the constant `the_empty_list` have been defined. For stacks, the types `stack_type` and `stack_item_type`, and the functions `create_stack`, `push`, `pop`, and `empty_stack` have been defined. For queues, the types `queue_type` and `queue_item_type`, and the functions `create_queue`, `enqueue`, `dequeue`, and `empty_queue` have been defined. Unless stated otherwise, assume that `list_item_type`, `stack_item_type` and `queue_item_type` are all `int`. Functions that manipulate lists, stacks, or queues must do so using only these standard interfaces; do not assume a particular underlying representation for any of these data structures. Also assume that the type `BOOL` and the constants `TRUE` and `FALSE` have been defined.

7. Write a function called `find_sub` that finds the first substring of a string that satisfies a given predicate. `Find_sub` takes two arguments: a string (a null–terminated array of `char`s) and a predicate (which takes a string as its argument and returns a `BOOL`). `Find_sub` returns a string. For example, if `begins_with_vowel` is a predicate that returns `TRUE` if its argument begins with a vowel (*i.e.*, one of the letters a, e, i, o, or u), and `FALSE` otherwise, then the call `find_sub("grotesque", begins_with_vowel)` should return the substring `"otesque"`. If no substring of the given string satisfies the given predicate, `find_sub` should return `NULL`. For example, `find_sub("pdq", begins_with_vowel)` should return `NULL`.

8. Write a function called `maximize` that returns the integer argument that produces the maximum value of a function between particular bounds. `Maximize` takes three arguments: a function (which takes an `int` as its argument and returns another `int`), and lower and upper bounds between which to search for the maximum value of the function (both `int`s). `Maximize` returns that integer between the lower and upper bounds, inclusive, that maximizes the return value of the functional parameter. As an example, suppose that function `foo` is defined as:

```
int
foo(int arg)
{
  return((4 - x) * (2 + x));
}
```

   then the call `maximize(foo, -2, 2)` should return `1`, because `foo(1) = 9` (which is larger than the result that `foo` returns for any other number in the range `-2` through `2`). Pay particular attention to the fact that the return value in this example is not `9`.

9. Write a C function called `apply_to_larger` that takes a function and two integers as arguments, and returns the integer resulting from applying the function to the larger of the two integers. The functional parameter takes an integer as its argument and returns an integer.

10. Write a C function called `add_sillies`. This function will take three arguments:

    (a) an array of `int`s;

    (b) an `int` representing the number of values in the array; and

    (c) a function that takes an `int` as an argument and returns `TRUE` or `FALSE` depending upon whether that `int` is a silly `int`.

    Your function must add up all the silly numbers in the array, and return the result. You will determine whether an array value is silly by calling the functional parameter.

11. Write a function called `count_if` that counts the number of array elements that satisfy a particular predicate. `Count_if` takes three arguments: an array of integers, the number of integers in the array, and a predicate. The predicate takes an integer as its argument, and returns a BOOL.

12. Write a function called `print_roots` that prints those integer roots of a function that lie between certain bounds (a root of a function is simply a value which, when handed to the function as an argument, causes the function to return zero). `Print_roots` takes three arguments: a function (which takes an `int` as its argument and returns another `int`), and lower and upper bounds between which to search for the roots of the function (both `ints`). For each integer between the lower and upper bounds inclusive, `print_roots` determines whether that number is a root of the function; if it is, the root is printed. `Print_roots` returns nothing. As an example, suppose that function `foo` is defined as:

```
int
foo(int x)
{
  return(x * x + x - 6);
}
```

Then the call `print_roots(foo, -3, 3)` should print:

```
x = -3
x = 2
```

because `-3` and `2` are both roots of the function in the range `-3` to `3`.

13. Write a function called `count_pairs` that takes a list and a list item as arguments. `Count_pairs` returns the number of times that the specified item appears twice in a row in the list. For example, your function should exhibit the following behavior:

```
count_pairs(<1 2 17 17 3>, 17) ⇒ 1
count_pairs(<1 17 2 17 3>, 17) ⇒ 0
count_pairs(<1 17 17 17 2>, 17) ⇒ 2
```

Notice that a single list element might take part in two pairs, as in the last example above.

14. Write a function called `list_swap` that takes a `list_type` and two `list_item_types` as arguments. `List_swap` returns a new list that is identical to the original list, except that every occurrence of the first `list_item_type` has been replaced by the second, and every occurrence of the second `list_item_type` has been replaced by the first. For example, your function should exhibit the following behavior:

```
list_swap(<1 2 42 3 17 42 4>, 17, 42) ⇒ <1 2 17 3 42 17 4>
list_swap(<1 2 17 17 3>, 17, 42) ⇒ <1 2 42 42 3>
list_swap(<1 2 3>, 17, 42) ⇒ <1 2 3>
```

15. Write a recursive C function called `is_member` that takes a list item and a list as arguments, and returns `TRUE` if the item is a member of the list, and `FALSE` otherwise.

16. Write a recursive C function called `all_but_first_n` that takes a list and a non—negative integer n as arguments. `All_but_first_n` returns a list composed of those items in the list after the first n items. Assume that the first item in the list is number 1. If the integer provided is greater than the length of the list, `the_empty_list` is to be returned. For example, your function should exhibit the following behavior:

```
all_but_first_n(<1 2 3 4 5>, 2) ⇒ <3 4 5>
all_but_first_n(<1 2 3 4 5>, 5) ⇒ < >
all_but_first_n(<1 2 3 4 5>, 8) ⇒ < >
```

17. Write a recursive C function called `remove_first_occurrence` that takes a list and a list item as its arguments and returns a list composed of the items in the original list except for the first occurrence of the list item. If the item does not occur in the list, the return list should contain exactly the same items as the original list. For example, your function should exhibit the following behavior:

```
remove_first_occurrence(<1 2 3 4 2>, 2) ⇒ <1 3 4 2>
remove_first_occurrence(<1 2 3 4 2>, 4) ⇒ <1 2 3 2>
remove_first_occurrence(<1 2 3 4 2>, 8) ⇒ <1 2 3 4 2>
remove_first_occurrence(< >, 42) ⇒ < >
```

18. Write a function called `reduce_list` that takes a list of numbers as its argument and builds a new list made up of the sums of pairs of numbers in the original list. That is, the first element of the result should be the sum of the first and second elements of the argument; the second element of the result should be the sum of the third and fourth elements of the argument; *etc.* For example, your function should exhibit the following behavior:

    reduce_list(<1 2 3 4 5 6>) ⇒ <3 7 11>
    reduce_list(<1 2 3 4 5>) ⇒ <3 7 5>
    reduce_list(<1 2 3 4>) ⇒ <3 7>
    reduce_list(<1>) ⇒ <1>
    reduce_list(<>) ⇒ < >

    Notice that if there is an odd number of values in the original list, the last value is copied unaltered to the result.

19. Write a C function called `collect_odds`. Your function must take a `list_type` as its argument; it will return a list_type as its result that contains all of the odd numbers found in the input list, in the same order as they occurred in the original list.

20. Write a recursive C function called `max_truth` that returns the maximum list item for which a given condition holds. `max_truth` takes two arguments: a `list_type` and a predicate which takes one `list_item_type` as its argument and returns either `TRUE` or `FALSE`. `Max_truth` should return the largest element of the list for which the predicate returns `TRUE`. If the predicate does not return `TRUE` for any element of the list, `max_truth` should return the constant `FAIL` (which will not be a legal list element—assume that `FAIL` is a legal `list_item_type` that has already been defined and that will never be an element of the list). Here are some examples of the behavior that your function should exhibit. These examples assume that `is_odd` is a predicate that takes an `int` as its argument and returns `TRUE` if that `int` is an odd number and `FALSE` if it is even:

    max_truth(<1, 2, 3, 4, 5>, is_odd) ⇒ 5
    max_truth(<2, 4, 6, 8>, is_odd) ⇒ FAIL
    max_truth(< >, is_odd) ⇒ FAIL
    max_truth(<2, 4, 5, 3, 6>, is_odd) ⇒ 5
    max_truth(<-4, -7, -9>, is_odd) ⇒ -7

21. Write the function `copy_queue` that takes a `queue_type` as its argument and returns another `queue_type` as its result. The return queue must contain all of the items in the original queue in the same order that they were found in the original queue. The original queue must end up unchanged.

22. Write a function called `reverse_queue` that reverses the order of the items in a queue. `Reverse_queue` is a `void` function that takes a `queue_type` as its argument. `Reverse_queue` must use a local variable of type `stack_type` to accomplish the reversal. Assume that `stack_item_type` and `queue_item_type` are the same type.

23. Write a function called `lift_item` that moves the top–most occurrence of a given value to the top of a stack. `Lift_item` takes two arguments: an item to lift of type `stack_item_type`, and a `stack_type`. `Lift_item` returns nothing. With the exception of the item lifted, `lift_item` should not affect the stack. Thus, after a call to `lift_item`, the stack should have the same number of arguments, and those arguments should be (with the exception of the item lifted) in the same order. If the stack contains no occurrences of the given item, it should be unchanged by a call to `lift_item`. If the stack contains more than one occurrence of the given item, only the top–most occurrence should be lifted.

24. Write a C function called `stack_extract` that takes a stack and a stack item as arguments, and removes the top–most occurrence of the item from the stack. The remaining items on the stack must be left in their original order. If the given item does not appear in the stack, nothing should be removed from the stack.

25. Write a C function called `bottom_item` that takes a stack as its argument and returns the item at the bottom of the stack. `Bottom_item` must leave the stack unchanged. You should use an `assert` statement to generate an error if `bottom_item` is handed an empty stack as its argument.

26. Write a C function called `remove_if` that takes a stack and a predicate, and removes all elements from the stack that satisfy the predicate. The predicate will be a function that takes a `stack_item_type` as its argument and returns a `BOOL`. Items that do not satisfy the predicate must remain on the stack in their original order.

27. A *foople* is an abstract data type that holds a collection of items. The following types, operations, and constants are defined for fooples:

    - `foople_type`: the type of a foople.
    - `foople_item_type`: the type of the objects contained in a foople.
    - `the_empty_foople`: a constant representing the foople that contains no objects.
    - `BOOL item_count(foople_type foople, int count)`: a predicate that returns `TRUE` if `foople` contains exactly `count` items, and returns `FALSE` otherwise.
    - `foople_type augment_foople(foople_type foople, foople_item_type item)`: a constructor that takes a foople and an item as arguments, and returns a new foople that contains all of the items in the original foople plus the new item. `Augment_foople` does not modify its arguments in any way. `Augment_foople` properly handles duplicate items.
    - `foople_item_type random_item(foople_type foople)`: a selector that returns one of the items in the argument foople at random. `Random_item` is non–destructive; it does not modify its argument in any way.
    - `foople_type nifty_items(foople_type foople)`: a selector that returns all of the items in the argument foople that are considered to be *'nifty.'* Because `nifty_items` returns a new foople, you might also consider it to be a constructor function. `Nifty_items` does not modify its argument in any way.
    - `foople_type nasty_items(foople_type foople)`: a selector that returns all of the items in the argument foople that are considered to be *'nasty.'* `Nasty_items` does not modify its argument in any way.

    Important notes about nifty items and nasty items:

    - Every item in a foople is considered to be either nifty or nasty.
    - An item that is considered to be nifty when it is in one foople might be considered to be nasty in another foople.
    - Every foople that contains two or more items contains at least one nifty item and at least one nasty item (*i.e.,* no foople with more than one item is made up of all nifty items or of all nasty items).

    You are to write a short recursive C function called `merge_fooples`. `Merge_fooples` takes two `foople_type`s as arguments and returns a new `foople_type` that contains all of the items in the first argument, and all of the items in the second argument. You may assume that all of the constants, types, and operations described above (including `BOOL`, `TRUE`, and `FALSE`) have already been defined.