Final Exam

CSC 2/454

19 December 2013

Directions; PLEASE READ

This exam comprises a mixture of short-answer and problem/essay questions. Values are indicated for each; they total 100 points. Questions 23 and 24 are worth up to 13 extra credit points; they won’t factor into your exam score, but may help to raise your letter grade when I calculate final grades.

This is a closed-book exam. You must put away all books and notes. Please confine your answers to the space provided. In the interest of fairness, I will generally decline to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You will have a maximum of three hours to complete the exam, though hopefully you won’t need it all. Any remaining exams will be collected promptly at 10:30 pm. Good luck!

A Freebie

1. (2 points) Put your name on every page (so if I lose a staple I won’t lose your answers).

Multiple Choice (3 points each)

2. Which of the following is not considered an essential feature of object-oriented programming?
   - a. encapsulation
   - b. generics
   - c. dynamic method dispatch
   - d. inheritance

3. Why are in-line functions useful?
   - a. They eliminate subroutine-call linkage overhead.
   - b. They enable the compiler to perform more code improvement.
   - c. They may improve I-cache locality.
   - d. All of the above.

4. Why might a compiler choose not to in-line a function?
   - a. It may be logically impossible.
   - b. It may worsen I-cache locality.
   - c. It may increase code size.
   - d. All of the above.
5. Why do most languages not specify the order of evaluation of arguments?
   - a. Because it doesn’t change the meaning of programs.
   - b. Because it’s already constrained by the precedence and associativity rules.
   - c. Because many important code improvement techniques depend on being able to change the order.
   - d. Because the order can’t be specified with context-free rules.

6. Which of the following is least likely to be used to generate code for a `switch` statement?
   - a. hash table
   - b. linear sequence of tests
   - c. balanced search tree
   - d. binary search in an array

7. Why does OCaml require that all elements of a list have the same type (in contrast to Scheme, which allows them to have different types)?
   - a. So that the language implementation can perform all type checks at compile time.
   - b. Because the language designers believed that heterogeneous lists were bad programming style.
   - c. To eliminate the need for generics.
   - d. Because heterogeneous lists are incompatible with OCaml’s richer syntax (in contrast to Scheme, where everything looks like a parenthesized list anyway.)

8. In which of the following situations is a spin lock an appropriate mechanism for mutual exclusion?
   - a. Synchronization among threads of a scientific simulation, running on a parallel supercomputer.
   - b. Synchronization between a graphical program and its signal handlers.
   - c. Synchronization between processes connected by a Unix pipe (as in `grep foo *.c | less`).
   - d. Control of access to a critical section that writes data to a shared file.

**Short Answer** (4 points each)

9. Why is the distinction between mutable and immutable values important in a language with a reference model of variables?
   **Answer:** Because the compiler can safely create copies of immutable values, and avoid the overhead of indirection.

10. A Java compiler will reject, at compile time, all programs that use an uninitialized variable. How can it do this without solving the halting problem?
    **Answer:** By also rejecting some programs that do not use an uninitialized variable.
11. Why doesn’t a program run any faster if you leave out optional parameters in subroutine calls?

**Answer:** Because the compiler inserts the missing parameters for you; the work performed at run time is exactly the same.

12. Ada 95 permits the programmer to create pointers to objects that do not lie in the heap, so long as those objects have been declared to be “aliased.” Why do you suppose the language designers required the special declaration?

**Answer:** So the implementation can use locks and keys, which require extra space in any object referred to by a pointer.

13. Older compilers often pushed subroutine arguments onto the stack one at a time, using instructions that automatically updated the stack pointer on each push. Modern compilers typically preallocate the space for in-stack arguments, and store into that space using static offsets from the stack pointer. Why the change?

**Answer:** So the arguments can be computed (and stored) out of order, if the compiler’s analysis suggests that this will result in faster code.

14. Why do some languages require the step size of a for loop to be a compile-time constant?

**Answer:** So the direction of the end-of-loop test (\(<\) or \(>\)) can be statically determined.

15. How can a compiler accommodate step sizes that are not known at compile time?

**Answer:** Don’t test the index to detect termination; rather, calculate the number of iterations in advance, count down one per iteration, and stop when you reach zero.

**Discussion / Problem Solving**

16. (6 points) Consider an implementation of C on a machine with 8-byte **doubles**, 2-byte **shorts**, and 1-byte **chars**, each of which must be stored in memory at an address that is an even multiple of its size. Then consider the following declarations:

```c
struct foo {
    double d;
    char c;
    short s;
};
struct foo v;
struct foo A[10];
```

Explain the layout of object `v` in memory. What is `v`’s size in bytes? What is the offset of each field? What is `A`’s size?

**Answer:** Object `v` comprises 8 bytes for `d`, followed by 1 byte for `c`, followed by 1 byte of garbage (a hole), 2 bytes for `s`, and 4 more bytes of garbage—a total of 16 bytes, with `d`, `c`, and `s` at offsets 0, 8, and 10, respectively. The need for the second hole can be seen when laying out `A`, which is 160 bytes long: without the second hole in `A[0]`, the `d` field of `A[1]` would not be properly aligned.
17. (8 points) Consider two global arrays, \( A \) and \( B \), declared as follows in C:

\[
\begin{align*}
\text{struct foo} & \quad A[10][10] = \ldots \\
\text{struct foo} & \quad *B[10] = \ldots
\end{align*}
\]

where \text{struct foo} is as in the previous question. Clearly \( A \) has contiguous layout, while \( B \) has row-pointer layout. Now consider the assignments

\[
\begin{align*}
double \ x & = A[i][3].d; \\
double \ y & = B[i][3].d;
\end{align*}
\]

where \( i \) is a variable whose value cannot be statically determined, but which has already been loaded into a register at run time. Describe the sequence of operations required to load the right-hand side of each assignment into a register. In particular, indicate for each right-hand side the number of addition, multiplication, and load operations required. You may assume that the machine provides scaled indexing, so \( w + z \times p \) can be calculated with a single addition instruction, if \( p \) is the size of a pointer.

**Answer:** To load \( A[i][3].d \), we need to compute \( &A + i \times 10 \times s + 3 \times s = [\&A + 3 \times s] + (i \times t) \), where \( s \) is the size of \text{struct foo} and \( t = 10 \times s \). The expression in square brackets is a compile-time constant, so our computation takes one multiplication, one addition, and a load.

To load \( B[i][3].d \), we need to compute \( *(\&B + (i \times p)) + (3 \times s) \), where \( s \) is again the size of \text{struct foo}, \( p \) is the size of a pointer, and \( [3 \times s] \) is a compile-time constant. Our computation requires one (scaled) addition, a load, a second addition, and a second load.

18. (6 points) When a parameter is passed by reference, the formal parameter is an alias for the actual parameter. When a parameter is passed by value/result, the formal parameter is initialized with a copy of the actual parameter at subroutine start-up, and the actual parameter is assigned a copy of the formal parameter at subroutine completion. (a) Briefly summarize the semantic and implementation tradeoffs between these two parameter passing modes. (b) Describe a simple program that would behave differently depending on whether a certain parameter is passed by reference or by value/result.

**Answer:** (a) Pass-by-value/result is generally considered to have cleaner semantics than pass-by-reference. The aliases created by reference parameters can lead to subtle bugs, and can make it difficult for a compiler to apply certain important performance-enhancing code optimizations. The usual implementation of pass-by-value/result is generally faster than the usual implementation of pass-by-reference for small (e.g., one-word) parameters. Pass-by-reference is generally faster for large (i.e. multi-word) parameters. Pass-by-reference takes less space for large parameters. (b) A simple example program that can tell the difference between the modes passes a global variable to a subroutine that modifies the corresponding formal parameter and then reads the global variable. Alternatively, any program that raises an exception in the middle of a subroutine may see different effects for value/result and reference parameters, since the copy-back of result parameters will not occur if the exception propagates out of the subroutine, but reference actual parameters may already have been modified in-place.
19. (8 points) Consider a parallel program that makes use of a partition abstraction. The partition comprises a collection of objects and buckets, and maintains the invariant that a given object lies in exactly one bucket at any given point in time. A move method of the abstraction serves to change the bucket in which an object lies (i.e., to move it from one bucket to another), atomically.

One possible implementation uses a single lock to protect the entire partition. Another uses a separate lock for every bucket. Discuss the tradeoffs between these.

**Answer:** A single lock is easy to code and will work quite well so long as the number of threads contending for simultaneous access is not large enough to make the partition a bottleneck. Multiple locks will permit a higher degree of concurrency, but they incur higher overhead in the no-contention case, since two of them will have to be acquired and released by each invocation of move. Moreover, it may be tricky to write the multiple-lock version correctly: if one thread tries to lock buckets A and B, in that order, while another tries to lock B and A, in that order, deadlock may result.

20. Consider the following context-free grammar:

```
expr → num | id | fn_call
fn_call → id ( args )
args → expr more_args | ε
more_args → , expr more_args | ε
```

(a) (4 points) Prove that this grammar is not LL(1).

**Answer:** id ∈ [predict(expr → id) ∪ predict(expr → fn_call)].

(b) (4 points) Show how to modify it to make it LL(1).

**Answer:**

```
expr → num | id id_tail
id_tail → ( args ) | ε
args → expr more_args | ε
more_args → , expr more_args | ε
```

21. As part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the following code:

```c
typedef struct list_node {
    void* data;
    struct list_node* next;
} list_node;

list_node* reverse(list_node* L) {
    list_node* rtn = 0;
    while (L) {
        rtn = insert(L->data, rtn);
        L = L->next;
    }
    return rtn;
}

list_node* insert(void* d, list_node* L) {
    list_node* t = (list_node*) malloc(sizeof(list_node));
    t->data = d;
    t->next = L;
    return t;
}

void delete_list(list_node* L) {
    while (L) {
        list_node* t = L;
        L = L->next;
        free(t->data);
        free(t);
    }
}
```
(a) (4 points) Accustomed to Java, new team member Brad includes the following code in the main loop of his program:

```c
list_node* L = 0;
while (more_widgets()) {
    L = insert(next_widget(), L);
}
L = reverse(L);
```

Sadly, after running for a while, Brad’s program always runs out of memory and crashes. Explain what’s going wrong.

**Answer:** The `reverse_list` routine produces a new list, composed of new list nodes. When Brad assigns the return value back into `L` he loses track of the old list nodes, and never reclaims them. In other words, his program has a memory leak. After some number of iterations of his main loop, Brad has exhausted the heap and his program can’t continue.

(b) (4 points) After Janet patiently explains the problem to him, Brad gives it another try:

```c
list_node* L = 0;
while (more_widgets()) {
    L = insert(next_widget(), L);
}
list_node* T = reverse(L);
delete_list(L);
L = T;
```

This seems to solve the insufficient memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

**Answer:** While the call to `delete_list` successfully reclaims the old list nodes, it also reclaims the widgets. The new, reversed list thus contains dangling references. These refer to locations in the heap that may be used for newly allocated data, which may be corrupted by uses of the elements in the reversed list. Brad seems to have been lucky in that he isn’t corrupting the heap itself (maybe his widgets are the same size as list nodes), but without Janet’s help he may have a lot of trouble figuring out why widgets are changing value “spontaneously.”

22. (5 points) Explain why the following OCaml function is difficult to translate into C.

```ocaml
let plus_n n = fun x -> n + x;;
```

**Answer:** First, it employs a lambda expression, which C lacks. Second, it requires `n` to have unlimited extent, which C also lacks.

23. (Extra Credit: up to 5 points) C++ allows one to overload “operator(),” the function-call notation. If `f` is an object of a class with an `operator()` method, then `f(x)` is syntactic sugar for `f.operator()(x)`. Explain how this mechanism can be used to approximate the behavior of the `plus_n` function of the previous question. How good is the approximation?
Answer:

class plus_n {
    int _n;
public:
    plus_n(int n) {_n = n;}
    int operator()(int x) {return _n + x;}
};

int main() {
    plus_n& plus3 = *(new plus_n(3));
    cout << plus3(5) << endl;  // prints 8
}

Once created, plus3 works as well as it would if created with let plus3 = plus_n 3 in OCaml. The creation mechanism is quite a bit more clumsy, however, which helps to explain why the standards committee chose to add lambda expressions to C++11. (Sadly, they did not add unlimited extent.)

24. (Extra Credit: up to 8 points) The concurrent functional language Multilisp is a dialect of Scheme with one additional built-in functional form, called future. Any expression in Multilisp can be embedded inside a future:

(future old-expression)

The future arranges for its embedded expression to be evaluated by a separate thread of control. The parent thread continues to execute until it actually tries to use the expression’s value, at which point it waits for execution of the future to complete. If two or more arguments to a function are enclosed in futures, then evaluation of the arguments can proceed in parallel:

(parent-func (future arg-expr-1) (future arg-expr-2))

Discuss the circumstances under which a Scheme programmer can safely and profitably insert futures into a program. Hint: consider the use of functional and imperative language features. Also consider the overhead of thread creation, scheduling, and synchronization.

Answer: In a purely functional program, future is semantically neutral: program behavior will be exactly the same as if the embedded expression had appeared without the surrounding call. In a program that uses imperative features of Scheme, the Multilisp programmer must ensure that there are no races between the argument to a future and any other code that might be executed in parallel.

Even in code that is purely functional, the programmer should consider whether an expression is complicated enough to warrant evaluation by a separate thread. Passing too small an expression to a future can result in a loss in performance, if the parent thread spends more time forking and joining with the child than it would have spent evaluating the expression itself. Passing an expression to a future also makes no sense unless a “sibling” argument is also made a future, or unless the parent will not need the value for a while.