Final Exam (corrected)

CSC 2/454

16 December 2015

Directions; PLEASE READ

This exam comprises 33 multiple-choice questions and 2 essay-style extra-credit questions. Multiple-choice questions 6 through 9 are worth one point each; the others are worth two points each. You also get 3 points for putting your name on every page, for a total of 62 points.

The extra credit questions, together, are worth up to 10 additional points; they won't factor into your exam score, but they may help to raise your end-of-semester letter grade.

This is a *closed-book* exam: you must put away all books, cellphones, and notes. Please confine your answers to the space provided. For multiple choice questions, darken the circle next to the single best answer. Be sure to read all candidate answers before choosing. No partial credit will be given on the multiple-choice questions.

In the interest of fairness, the proctor has been instructed not to try to explain anything on 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 3 hours to complete the exam, though it should take substantially less than that. The proctor will collect any remaining exams promptly at 7:00 pm. Good luck!

1. (**required**) Per college policy, please write out the following statement and add your signature: "I affirm that I will not give or receive any unauthorized help on this exam, and that all work will be my own."

   Signature: ________________________________

2. (3 points) Put your name on each of the remaining pages (so if I lose a staple I won't lose your answers).

Multiple Choice

3. The grammars used in many production compilers contain *error productions*, which allow the parser to accept certain input programs that are, according to the manual, syntactically invalid. What purpose do these error productions serve?

◯ **a.** They improve compilation speed by catching common errors more quickly than a general-purpose error recovery scheme would.

◯ **b.** They support language features that can be parsed bottom-up, but not top-down.

◯ **c.** They provide more helpful error messages than a general-purpose error recovery scheme would.

◯ **d.** none of the above

4. Consider the following program fragment, written in no particular language:

```
string tag = "b"    // global declaration

function print_formatted_string(string s)
    print "<", tag, ">", s, "</", tag, ">"

function emphasize (string s)
    string tag = "i"
    print_formatted_string(s)

begin               // main program
    emphasize("hi, mom")
```

What does this program print?

◯ **a.** It prints “`<b>hi, mom</b>`” if the language uses static (lexical) scoping, and “`<i>hi, mom</i>`” if the language uses dynamic scoping.

◯ **b.** It prints “`<b>hi, mom</b>`” if the language uses dynamic scoping, and “`<i>hi, mom</i>`” if the language uses static (lexical) scoping.

◯ **c.** It prints “`<b>hi, mom</b>`” regardless of scope rules.

◯ **d.** It prints “`<i>hi, mom</i>`” regardless of scope rules.

5. Why do most languages not allow the bounds or increment of an enumeration-controlled (`for`) loop to be floating-point numbers?

◯ **a.** Because the number of iterations could depend on round-off errors, leading to unpredictable behavior.

◯ **b.** Because floating-point arithmetic is so much more expensive than integer arithmetic.

◯ **c.** Because loop control instructions would have to be executed in the floating point unit, and communication between the integer and floating-point ALUs would slow the pipeline down.

◯ **d.** Because floating point numbers cover a dramatically wider range of values, and loops might run too long.

The next four questions are worth one point each. Characterize the polymorphism in each example.

6. In C:

```
a = b * c;      /* integer or floating point multiplication? */
```

◯ **a.** ad hoc polymorphism (overloading)
◯ **b.** subtype polymorphism
◯ **c.** explicit parametric polymorphism
◯ **d.** implicit parametric polymorphism

7. In Java:

```
Iterator it = my_GUI_set.iterator();
while (iterator.hasNext()) {
     Object elem = it.next();
     elem.display();      // which "display" method?
}
```

◯ **a.** ad hoc polymorphism (overloading)
◯ **b.** subtype polymorphism
◯ **c.** explicit parametric polymorphism
◯ **d.** implicit parametric polymorphism

8. In C++:

```
stack<int> int_stack;
stack<double> double_stack;      // both use library stack abstraction
int_stack.push(3);
double_stack.push(3.14159);
```

◯ **a.** ad hoc polymorphism (overloading)
◯ **b.** subtype polymorphism
◯ **c.** explicit parametric polymorphism
◯ **d.** implicit parametric polymorphism

9. In Scheme:

```
(define sort (L) ...
(sort '("these" "are" "a" "bunch" "of" "strings"))
(sort '(1 3 9 4 2 5 8)) ; and these are a bunch of ints
```

◯ **a.** ad hoc polymorphism (overloading)
◯ **b.** subtype polymorphism
◯ **c.** explicit parametric polymorphism
◯ **d.** implicit parametric polymorphism

10. In C++, the `static_cast` operator can be used to perform an explicit, unchecked type conversion. It is roughly equivalent to the parenthesized typename "casts" of C. The compiler will generate code to convert values when appropriate. The `dynamic_cast` operator can be used to convert pointers "the wrong way" in a polymorphic class hierarchy. The compiler will generate code to perform a run-time check of the concrete type of the referred-to object. The `reinterpret_cast` operator can be used to look at the underlying bits of an object as if they represented an object of a different type.

    Which of these three operators is type-safe?

    ◯ **a.** `static_cast`
    ◯ **b.** `dynamic_cast`
    ◯ **c.** `reinterpret_cast`
    ◯ **d.** none of the above; all are unsafe

11. Consider the following C++ code fragment:

```
int a = 3;
int b = 2;
double r = a/b;
```

    Suppose we want `r` to have the value `1.5`, rather than `1`. Which cast operator can be used in the third line to achieve this effect?

    ◯ **a.** `static_cast`
    ◯ **b.** `dynamic_cast`
    ◯ **c.** `reinterpret_cast`
    ◯ **d.** none of the above

12. Why won't C++ allow you to apply a `dynamic_cast` operator to an object of a class that has no virtual functions?

    ◯ **a.** Because there is no reason one would ever *need* to convert "the wrong way" in a class hierarchy without virtual functions.
    ◯ **b.** Because objects without virtual functions have no vtables, and the language implementation has no way to determine their concrete type at run time.
    ◯ **c.** Because the `dynamic_cast` operator itself is implemented with virtual functions.
    ◯ **d.** Because the whole point of objects without virtual functions is to minimize run-time costs, and `dynamic_cast` *imposes* run-time costs.

13. Suppose we are compiling for a machine with 1-byte characters, 2-byte shorts, 4-byte integers, and 8-byte reals, and with alignment rules that require the address of every primitive data element to be an integer multiple of the element's size. Suppose further that the compiler is not permitted to reorder fields. How much space will be consumed by the following array?

```
A : array [0..9] of record
    s : short;
    c : char;
    t : short;
    d : char;
    r : real;
    i : integer;
end;
```

- ◯ **a.** 150 bytes
- ◯ **b.** 320 bytes
- ◯ **c.** 240 bytes
- ◯ **d.** 200 bytes

14. 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

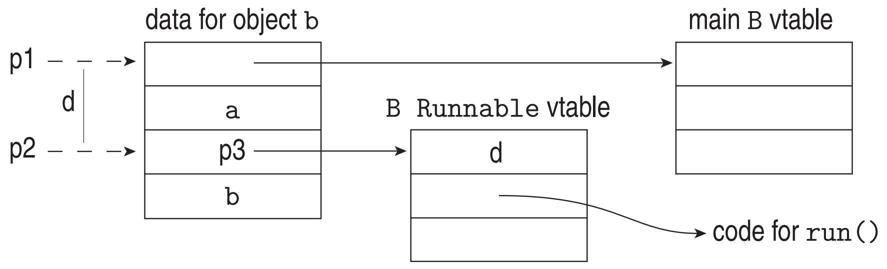
15. Consider a C program that contains a `switch` statement with 8 arms, labeled with the values 1, 2, 3, 4, 5, 7, 8, and 9 (no 6, no `default`). Suppose that our compiler has decided to implement the statement with a characteristic array (jump table). What should it put in the 6th entry of the table?

- ◯ **a.** a pointer to the code for the 7 case, since that is the sixth alternative
- ◯ **b.** a pointer to the code for the 5 case, since 6 is between 5 and 7, and `switch` statements in C fall through to the next case by default
- ◯ **c.** a pointer to code that announces a run-time error
- ◯ **d.** a pointer to the code that comes after the `switch` statement

The next 2 questions assume the following Java code:

```
class A {
    private int a;
    ...
}
class B extends A implements Runnable {
    private int b;
    ...
}
...
ExecutorService pool = Executors.newCachedThreadPool();
...
B b = new B();
...
pool.execute(b);
...
```

Recall that `Runnable` is a library interface that defines a zero-argument `run` method. `ExecutorService` is also a library interface; it defines an `execute` method that takes a single `Runnable` argument. You may assume that the compiler lays out data structures as follows:



16. What value will the compiler pass as the explicit parameter to `execute`?

   ◯ **a.** p1
   ◯ **b.** p2
   ◯ **c.** p3
   ◯ **d.** d

17. What value will `execute` pass to `run` as its implicit `this` parameter?

   ◯ **a.** p1
   ◯ **b.** p2
   ◯ **c.** p3
   ◯ **d.** d

18. Consider the following code in some unspecified language, where `a`, `b`, `c`, `x`, and `y` are distinct local variables of floating-point type:

```
x = (c + a) + b
y = (d + a) + b
```

Here are two possible translations into assembly language:

```
r1 = a            r1 = a
r2 = b            r1 += b
r3 = c            r3 = c
r3 += r1          r3 += r1
r3 += r2          x = r3
x = r3            r3 = d
r3 = d            r3 += r1
r3 += r1          y = r3
r3 += r2
y = r3
```

Most compilers will generate the version on the left, even though it's longer. Why?

◯ **a.** Most compilers are not smart enough to recognize that the two uses of `(a + b)` are redundant.

◯ **b.** The sum of `a` and `b` might overflow, leading to incorrect results.

◯ **c.** The version on the left is likely to have better cache behavior.

◯ **d.** The store to `x` might change the value of `a` or `b`.

19. Consider the following code sequences in C++, where `foo` is a user-defined class type:

```
foo a, b;                     foo a;
b = a;                        foo b = a;
```

Are these two sequences (the one on the left and the one on the right) guaranteed to produce the same behavior?

◯ **a.** Yes: the one on the right is just syntactic sugar for the one on the left.

◯ **b.** No: `foo`'s default constructor and its assignment operator may exhibit different behavior.

◯ **c.** No: the code on the right may access `a` or `b` before it has been initialized.

◯ **d.** No: the code on the left may execute the `a` and `b` constructors in either order, and they may have side effects.

20. Consider the following code in an unspecified language, where `a`, `b`, and `c` are distinct integer variables:

```
for i := a to b by c
    // body
```

Here are two possible translations into assembly language:

```
                                 r1 = a
    r1 = a                       r2 = ⌊(b − a + c) / c⌋
    goto L2                      is r2 ≤ 0 ?
L1: // body                      if so, goto L3
    r1 += c                  L1: // body
L2: is r1 ≤ b ?                  r1 += c
    if so, goto L1           L2: if --r2 > 0 goto L1
                             L3:
```

The second line on the right will require multiple instructions; every other line is a single instruction. Why might the compiler prefer the version on the right?

◯ **a.** It doesn't depend on `c` being positive.
◯ **b.** It doesn't depend on the body of the loop leaving `b` unchanged.
◯ **c.** It's likely to be faster.
◯ **d.** all of the above

The next 2 questions assume the following OCaml code:

```
let rec foo n l =
  match (n, l) with
  | (1, h :: t) -> Some h
  | (_, h :: t) -> foo (n - 1) t
  | _ -> None ;;
```

21. What is the type of `foo`?

◯ **a.** `int -> 'a list -> 'a option`
◯ **b.** `'a * 'b list -> 'b list`
◯ **c.** `int * int list -> int option`
◯ **d.** `'a -> 'b list -> 'b option`

22. What does `foo` *do*?

◯ **a.** It counts the number of elements in a given list.
◯ **b.** It returns the last `n` elements of a given list.
◯ **c.** It reverses the first `n` elements of a given list.
◯ **d.** It returns the `n`th element of a given list, if there is such an element.

23. Consider the following programs in Java (left) and C++ (right):

```
class A {
    public void foo() {
        System.out.println("A");
    }
}
class B extends A {
    public void foo() {
        System.out.println("B");
    }
}
class Dispatch {
    public static void main(String[] args) {
        A obj = new B();
        obj.foo();
    }
}
```

```
#include <iostream>
struct A {
    void foo() {
        std::cout << "A\n";
    }
};
struct B : A {
    void foo() {
        std::cout << "B\n";
    }
};
int main() {
    A* obj = new B();
    obj->foo();
}
```

What do these two programs print?

◯ **a.** `A` in Java, `B` in C++
◯ **b.** `A` in both
◯ **c.** `B` in Java, `A` in C++
◯ **d.** `B` in both

24. In many cases, local variables and other stack data can be found using offsets from the stack pointer (`sp`) register. Compilers often employ a separate frame pointer (`fp`), for this purpose, however, even on 32-bit x86 machines, where registers are an extremely scarce resource. What is the *most compelling* reason for having a separate `fp`?

◯ **a.** to support dynamic arrays
◯ **b.** to support displacement addressing with limited offsets
◯ **c.** to avoid the complexity of changing offsets when pushing and popping arguments
◯ **d.** to distinguish between local variables (at negative offsets) and parameters (at positive offsets)

25. Recall that *call-by-name* parameters are implemented as *thunks* (parameter-less subroutines) that can be called to force evaluation. *Call-by-need* parameters add a *memo* to hold the value once it is evaluated, so the thunk never has to be called more than once. Under what circumstances may call-by-name and call-by-need yield different results?

◯ **a.** when there isn't enough room to hold the memo in the current stack frame
◯ **b.** when the thunk may raise an exception
◯ **c.** when the thunk depends on data that is modified by the called routine, or by another thread
◯ **d.** when the value of the parameter is never actually needed

26. What is the principal insight behind *generational* garbage collection?

◯ **a.** Garbage collection is most likely to be productive at the end of some logical program phase.

◯ **b.** Many programs operate in a "pipelined" fashion; the objects most likely to be garbage are those created long ago.

◯ **c.** Most dynamically created objects don't live very long.

◯ **d.** Fragmentation can be reduced by copying objects to new, adjacent locations during garbage collection.

27. Why is garbage collection difficult in C?

◯ **a.** Because C was designed for systems programming, and can't tolerate hidden fields in objects.

◯ **b.** Because C is often used for real-time programming, and can't tolerate performance "hiccups."

◯ **c.** Because pointers can be created not only to objects in the heap, but to static and stack objects as well.

◯ **d.** Because the language isn't type-safe; we don't know where all the pointers are.

28. C++11 provides so-called *smart pointers*, which overload the various dereference and assignment operators to augment the functionality of pointers with automatic storage management. In particular, objects of class `std::shared_ptr` implement automatic reference counting. All `shared_ptr`s that refer logically to the same object O are implemented as pointers to a hidden, dynamically allocated object (essentially, a tombstone) that in turn contains a pointer to O. Why do you suppose the implementation needs this extra level of indirection?

◯ **a.** to avoid the need to reserve space for a reference count in every heap object

◯ **b.** to prevent the creation of circular references

◯ **c.** to catch dangling references

◯ **d.** to allow O itself to be copied to a new location, thereby reducing fragmentation

29. Which of the following is *not* true of generics (explicit parametric polymorphism) in Java?

◯ **a.** They can be type-checked once, at declaration time, ensuring that no additional errors will be announced when an instance of the generic is instantiated.

◯ **b.** They allow a single copy of the code for a generic to be shared among all instantiations.

◯ **c.** They avoid the need to put explicit type checks into one's source code when removing objects from a generic container.

◯ **d.** They avoid the run-time cost of type checks when removing objects from a generic container.

30. When using condition synchronization in Java (and most other modern languages), one must generally write

```
while (! desired_condition) {
    wait()
}
```

rather than

```
if (! desired_condition) {
    wait()
}
```

Why must one check the condition again after waiting?

◯ **a.** Because some other thread may have executed since the corresponding `notify`, and may have made the condition false again.

◯ **b.** Because other threads may call `notify` when the condition isn't actually true.

◯ **c.** Because the virtual machine is allowed to perform occasional spurious wakeups.

◯ **d.** all of the above

31. Consider the following Java code, where thread 1 and thread 2 execute in parallel:

```
// shared variables:
Object o;
volatile int a = 0;
int b = 0;
int c = 0;
volatile int d = 0;

// thread 1:                          // thread 2:
synchronized(o) {                     int x;  // local variable
    a++;                              synchronized(o) {
    b++;                                  x = b;
}                                     }
c++;                                  int y = a + x + c + d;
d++;
```

Which shared variable(s) are properly synchronized (free of data races)?

◯ **a.** variables `a`, `b`, and `d`

◯ **b.** variables `a` and `d` only

◯ **c.** variable `c` only

◯ **d.** variables `a` and `b` only

32. An *action routine*, as you may recall, is a semantic function together with an explicit indication of when it should be called during parsing. Action routines in a top-down context-free grammar constitute an ad hoc implementation of the semantic functions of an L-attributed attribute grammar. What data *cannot* accessed in such ad hoc routines?

   ◯ **a.** inherited attributes of the symbol on the left-hand side of the current production

   ◯ **b.** global variables

   ◯ **c.** synthesized attributes of symbols to the right of the action routine in the right-hand side of the current production

   ◯ **d.** none of the above—all are fair game in an action routine

33. Which of the following is *not* an accurate statement regarding the layout of multidimensional arrays?

   ◯ **a.** In the general case, access to an element of an array requires fewer multiplications with row-pointer layout than it does with contiguous layout.

   ◯ **b.** Row-pointer layout is less vulnerable to memory fragmentation, since it allocates the array in pieces.

   ◯ **c.** For an $N \times M$ array, row-pointer layout generally requires more space than contiguous layout.

   ◯ **d.** When one or more index is known at compile time, row-pointer layout leads to faster code than contiguous layout, because more of the address calculation can be pre-computed.

34. Dope vectors (run-time descriptors) are needed for arrays that

   ◯ **a.** are allocated in the heap instead of the stack.

   ◯ **b.** use row-pointer layout.

   ◯ **c.** have statically unknown bounds.

   ◯ **d.** contain elements of multiple types.

35. On 64-bit x86 processors, the stack pointer (`sp`) is said to protect not only all data below it (i.e., at higher addresses), but a 128-byte "red zone" of data *above* it as well (i.e., at lower addresses). If they require less than 128 bytes of stack frame, *leaf* routines (subroutines that call no other subroutines) can avoid updating the `sp`. What is the threat to data in the stack? Against what does the `sp` protect it?

   ◯ **a.** concurrent access in other threads

   ◯ **b.** iterators placed above the current stack frame

   ◯ **c.** arrays whose size increases during execution

   ◯ **d.** signal handlers

36. (**Extra Credit** up to 4 points) Discuss what you consider to be one of the most interesting interactions between language design and language implementation.

37. (**Extra Credit** up to 6 points) Discuss the interaction of exceptions and threads. Possible questions you might want to consider include: Does the existence of threads complicate the implementation of exceptions? Should one thread be able to explicitly raise an exception in another? What should happen if an exception escapes the outermost scope of a thread? Should an exception be allowed to escape a monitor?