

Final Exam

CSC 254

19 December 2006

Directions; PLEASE READ

This exam has 23 questions: 15 multiple-choice, 6 short-answer, and (for extra credit) 2 essays. The non-extra-credit questions total 100 points; values are indicated for each. (Several of the short-answer questions have multiple parts.) Questions 22 and 23 are worth up to 8 extra credit points each; they won't factor into your exam score, but may help to raise your end-of-semester letter grade.

This is a *closed-book* exam. You must put away all books and notes (a dictionary is permissible). Please confine your answers to the space provided. For multiple choice questions, darken the circle next to the best answer. Be sure to read all candidate answers before choosing.

Put your name on every page. That way if I lose a staple I won't lose your answers. (Please do this **NOW** so you don't forget.) No partial credit will be given on the multiple-choice questions.

In the interest of fairness, the proctor has been instructed not 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 I hope it won't take that long. The proctor will collect any remaining exams promptly at 11:30 am. Good luck!

Multiple Choice (3 points each)

1. What is the defining difference between a compiler and a preprocessor?

- a. A compiler produces machine language; a preprocessor produces source code.
- b. A compiler performs full syntactic and semantic analysis; a preprocessor does not.
- c. A compiler performs code improvement (optimization); a preprocessor does not.
- d. none of the above

2. Short-circuit evaluation of Boolean expressions

- a. may sometimes succeed in cases where regular (full) evaluation would result in a run-time error.
- b. will always lead to the same program behavior as regular evaluation evaluation, if both succeed.
- c. reduces the number of branch instructions in complicated expressions.
- d. all of the above

3. Which of the following errors will always be caught by a correct implementation of C++?
- a. array reference out of bounds
 - b. use of uninitialized variable
 - c. use of pointer to data that has already been freed (**deleted**)
 - d. none of the above
4. Which of the following is considered a *pure* functional language?
- a. Lisp
 - b. ML
 - c. Haskell
 - d. Scheme
5. Which of the following is true of attribute grammars?
- a. Every S-attributed grammar is also L-attributed.
 - b. A symbol in an attribute grammar can have synthesized attributes or inherited attributes, but not both.
 - c. L-attributed grammars can naturally be evaluated during a bottom-up (LR) parse.
 - d. Every attribute grammar also has a natural expression as a context-free grammar with action routines.
6. Why do most languages not specify the order of evaluation of arguments?
- a. Because many important code improvement techniques depend on being able to change the order.
 - b. Because it doesn't change the meaning of programs.
 - c. Because it's already constrained by the precedence and associativity rules.
 - d. Because the order can't be specified with context-free rules.
7. The *static link* in a stack frame indicates
- a. the frame of the calling routine.
 - b. the code of the calling routine.
 - c. the frame of the lexically-surrounding routine.
 - d. the base address for global variables.
8. Why does Java provide mix-in inheritance instead of full multiple inheritance?
- a. Because it avoids discontinuous objects.
 - b. Because it imposes no costs on programs that use only single inheritance.
 - c. Because it eliminates the distinction between replicated and shared repeated inheritance.
 - d. All of the above.

9. There are several reasons to prefer in-line functions over macros. Which of the following is *not* among these reasons?
- a. better opportunities for compiler optimization
 - b. more conventional parameter-passing semantics
 - c. more predictable behavior for arguments with side effects
 - d. use of a separate scope
10. *Closures*, which combine a subroutine pointer and a referencing environment, are required only
- a. for languages with static (lexical) scope.
 - b. for first-class subroutines.
 - c. for deep binding.
 - d. All of the above.
11. What is the value of `(car (cdr (map * '(1 2 3) '(4 5 6))))`?
- a. a run-time error
 - b. `()` (the empty list)
 - c. 120
 - d. 10
12. Under *name equivalence*, two variables have the same type if
- a. they have the same internal structure.
 - b. they were declared using the same lexical occurrence of a type constructor.
 - c. they can hold the same set of values.
 - d. they support the same set of methods.
13. Which of the following is most often employed by production-quality garbage collection systems?
- a. Mark-and-sweep
 - b. Generational collection
 - c. Tombstones
 - d. Locks and keys
14. What is the meaning of the Prolog statement `brillig(w) :- slithy(t), gyring(t,w).`?
- a. $\forall w[\text{brillig}(w) \rightarrow (\exists t[\text{slithy}(t) \wedge \text{gyring}(t,w)])]$
 - b. $\forall w[\text{brillig}(w) \leftarrow (\exists t[\text{slithy}(t) \wedge \text{gyring}(t,w)])]$
 - c. $\exists w[\text{brillig}(w) \rightarrow (\forall t[\text{slithy}(t) \wedge \text{gyring}(t,w)])]$
 - d. $\exists w[\text{brillig}(w) \leftarrow (\forall t[\text{slithy}(t) \wedge \text{gyring}(t,w)])]$

15. How might the following be written in Prolog? “If I forget my umbrella on a day when it rains, I’ll get wet.”

- a. `wet(m) :- rainy(d). wet(m) :- no_umbrella(m,d).`
- b. `rainy(d), no_umbrella(m,d) :- wet(m).`
- c. `rainy(d) :- wet(m). no_umbrella(m,d) :- wet(m).`
- d. `wet(m) :- rainy(d), no_umbrella(m,d).`

Short Answer

16. (a) (4 points) Describe in English the language defined by the following regular expression:

`a*(b a* b a*)*`

Please do *not* explain the regular expression. Describe the *language* in as intuitive a way as possible.

Answer: The set of all strings of `as` and `bs` containing an even number of `bs`.

(b) (6 points) Write an unambiguous context-free grammar that generates the same language.

Answer:

$$\begin{aligned}
 G &\rightarrow As S \\
 S &\rightarrow b As b As S \\
 &\rightarrow \\
 As &\rightarrow a As \\
 &\rightarrow
 \end{aligned}$$

Note that putting another `As` between the second `b` and the second `S` in the second production would make the grammar ambiguous.

(c) (6 points) Using your grammar from part (b), give a canonical (rightmost) derivation of the string `b a a b a a a b b`.

Answer:

$$\begin{aligned}
 &G \\
 &S As \\
 &S \\
 &As b As b S \\
 &As b As b As b As b S \\
 &As b As b As b As b \\
 &As b As b As b b \\
 &As b As b a As b b \\
 &As b As b a a As b b \\
 &As b As b a a a As b b \\
 &As b As b a a a b b \\
 &As b a As b a a a b b \\
 &As b a a As b a a a b b \\
 &As b a a b a a a b b \\
 &b a a b a a a b b
 \end{aligned}$$

```

class boundedBuffer {
    final private int SIZE = 10;
    private int nextEmpty = 0;
    private int nextFull = 0;
    private int fullSlots = 0;
    private Object[] buf = new Object[SIZE];    // circular queue

    private class condition {
        synchronized void doWait() {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        synchronized void doNotify() {
            notify();
        }
    }

    final private condition slotMadeFull = new condition();
    final private condition slotMadeEmpty = new condition();

    synchronized public void insert(Object d) {
        while (fullSlots == SIZE) {            // buffer is full
            slotMadeEmpty.doWait();           // wait for a consumer
        }
        buf[nextEmpty] = d;
        nextEmpty = (nextEmpty + 1) % SIZE;
        ++fullSlots;
        slotMadeFull.doNotify();             // wake up a consumer, if there is one
    }

    synchronized public Object remove() {
        Object d;
        while (fullSlots == 0) {                // buffer is empty
            slotMadeFull.doWait();             // wait for a producer
        }
        --fullSlots;
        d = buf[nextFull];
        nextFull = (nextFull + 1) % SIZE;
        slotMadeEmpty.doNotify();             // wake up a producer, if there is one
        return d;
    }
}

```

Figure 1: A broken implementation of a bounded buffer in Java.

17. (5 points) Consider the Java code of Figure 1. This code attempts to get around the one-condition-queue-per-monitor limitation of Java 2 by waiting on separate `condition` objects. It will not work correctly, however. Explain the synchronization bug.

Answer: When a thread waits in `condition.doWait` it will release the monitor lock on the condition itself, but it will retain the monitor lock on the bounded buffer. No other thread will be able to enter a buffer method to call `condition.doNotify`.

18. (a) (4 points) Several languages, including Ada, Python, and Ruby, define the meaning of declarations in terms of *elaboration*, a run-time process semantically similar to the execution of statements. Like statements, declarations achieve their effect one by one, sequentially, when control enters their scope during program execution. List at least four different things that may happen as a result of elaborating a declaration. Which of these require run-time code? Which can be accomplished statically by the compiler?

Answer: At compile time, elaboration may create a name-to-object binding; hide an existing name-to-object binding; assign to an object a static address, or an offset within a frame. At run time, elaboration may create an object and allocate space to hold it; check bounds (e.g., is first array index less than last?); raise exceptions if checks fail; assign an initial value to an object (or call a constructor); start execution of concurrent tasks.

- (b) (4 points) Explain why elaboration is a useful concept, *even for declarations that can be handled completely at compile time*.

Answer: From a semantic point of view, the distinction between compile time and run time is artificial—it's an artifact of the implementation. Elaboration couches everything in terms of execution—it allows the language definition to ignore the notion of compile time. Static decisions on the part of the compiler are merely optimizations.

19. Consider the following declaration in a Pascal-like language:

```
var r : record
    x : integer;
    y : char;
    A : array [1..10, 10..20] of record
        z : real;
        B : array [0..71] of char;
    end;
end;
var j, k : integer;
```

Assume that these declarations are local to the current subroutine. Assume further that the machine on which we are running has 4-byte integers, 8-byte floating-point numbers, 1-byte characters, and 4-byte alignment for integers and floating-point numbers. Finally, assume that the compiler uses contiguous row-major layout for multi-dimensional arrays, and does not pack or re-order fields of records. Note the lower bounds on indices in `A`; the first element is `A[1,10]`.

- (a) (4 points) Describe how `r` would be laid out in memory.

Answer: Variable `r` would lie in the stack frame of the current subroutine. It would contain, in order, 4 bytes for `x`, 1 byte for `y`, 3 bytes of hole, and 110 contiguous elements of `A`. Each element of `A` would contain 8 bytes of `z` followed by 72 bytes (an even multiple of 4) of `B`.

- (b) (6 points) Show pseudo-assembly code to access `r.A[2,j].B[k]`. Be sure to indicate which portions of the address calculation could be performed at compile time.

Answer: To access `r.A[2,j].B[k]`, one would perform the following calculation:

$$\begin{aligned} & \text{fp} && + \text{offset of } r \text{ in frame} \\ & && + \text{offset of } A \text{ in } r \text{ (i.e. } 8) \\ & && + (2-1) \times 11 \times \text{sizeof}(A[x,y]) \text{ (i.e. } 80) \\ + j \times \text{sizeof}(A[x,y]) & - 10 \times \text{sizeof}(A[x,y]) \\ + k \\ \\ = (\text{fp} + 80j + k) & + (\text{offset}(r) + 88) \end{aligned}$$

Values on the right are compile-time constants; values on the left are not known until run time. Suppose `offset(r)` is 12. Assembly code might look something like this:

```
r1 := j
r1 *= 80
r1 += k
r1 += 100      // offset(r) + 88
r1 := fp[r1]   // displacement addressing
```

20. 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) (5 points) Briefly summarize the semantic and implementation tradeoffs between pass-by-reference and pass-by-value/result.

Answer: 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) (5 points) Describe a program that would behave differently depending on whether a certain parameter is passed by reference or by value/result (you can give code if you want, but you don't have to).

Answer: 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.

21. (6 points) In a language with structured exception handling (`try/catch` blocks), the key implementation issue is how to implement the bookkeeping that allows us to find the right handler when an exception occurs. Briefly describe an implementation that imposes run-time cost only when exceptions actually arise (and *not* when entering and leaving a `try` block).

Answer: See page 450 in the text (*PLP* 2e).

Extra Credit

22. (8 points max) 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 `future`s, 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 `future`s 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.

23. (8 points max) As you may recall, the generics of Ada, C++, Java, C#, and others provide a form of explicit parametric polymorphism. Other languages, including Lisp, ML, Smalltalk, and most scripting languages, provide *implicit* parametric polymorphism: functions naturally accept arguments of any type that supports the operations applied to them by the function. Discuss the comparative strengths and weaknesses of explicit and implicit parametric polymorphism. For maximum points, illustrate your discussion with examples.

Answer: The implicit variety is arguably more flexible, and easier to read; the explicit is arguably safer, more precise, and potentially faster.

Implicit parametric polymorphism requires either Milner type inference (as in ML) or run-time checks (as in Scheme). Generics can be checked at compile time in most languages (though Java chose not to do so, for compatibility reasons), avoiding the run-time overhead. Because the compiler knows more about a routine with explicit type parameters, it may also be able to generate more efficient code. Certainly a C++ compiler could optimize the code to find the maximum element in a vector of `ints` in ways that an ML compiler or Common Lisp compiler would be unable to optimize general-purpose vector code. Such optimizations (*specializations*) clearly require multiple copies of the code; languages with implicit polymorphism use a single copy.