Directions—PLEASE READ

This exam comprises 20 multiple choice and 22 short answer questions. Each multiple choice question is worth 2 points, and each short answer question is worth 3 points, for a total of 106 points. There are also two essay-style questions that count for up to 6 extra credit points each.

This is a closed-book exam. You must put away all books and notes (except for a dictionary, if you want one). Please begin by putting your name on every page of the exam.

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 these questions. For the short answer questions, please answer in a single sentence. If you need two sentences you’re writing too much. Weak or overly verbose answers will receive only partial credit.

In the interest of fairness, the proctor will 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 work, though it shouldn’t take that long. The proctor will collect any remaining exams promptly at 10:15 pm. Good luck!

Multiple Choice

1. What is the significance of L-attributed attribute grammars?
   
   ☑ a. They are the most general class of attribute grammars that can be evaluated in linear time.
   
   ✗ b. They can be evaluated in the course of an LL parse.
   
   ☑ c. Their attribute flow is strictly bottom-up.
   
   ☑ d. They can be handled by the built-in evaluator in yacc/bison.

2. 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 the order can’t be specified with context-free rules.
   
   ✗ d. Because many important code improvement techniques depend on being able to change the order.

3. Why don’t you need a static link in C?
   
   ☑ a. Because a display is used instead.
   
   ☑ b. Because there’s already a separate frame pointer.
   
   ✗ c. Because C doesn’t have nested subroutines.
   
   ☑ d. Because it wouldn’t be type-safe.
4. In Ruby, an *iterator* is
   - a. A loop.
   - b. A tail-recursive function.
   - X c. A method of a collection that accepts a *block* as a parameter, and invokes the block once for every object in the collection.
   - d. A method that *yields* the elements of a collection, one at a time, for use in a separate *for* loop.

5. Why might one wish to pass a parameter by value/result, rather than by reference?
   - a. To avoid modifying the argument.
   - b. To save space in the stack.
   - c. To avoid the need for indirection.
   - d. None of the above.

6. Consider the following CFG:

   \[
   \text{list} \rightarrow \text{ids} ; \\
   \text{ids} \rightarrow \text{ids} , \text{id} \\
   \rightarrow \text{id}
   \]

   Which of the following is a sentential form for this language?

   - a. id, id, ids ;
   - X b. ids , id, id;
   - c. ids , ids ;
   - d. all of the above

7. 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.
   - X d. All of the above.

8. 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.
   - X d. All of the above.
Short Answer

9. What is (are) the initial source(s) of information in the decoration of a parse or syntax tree?
   Answer: Synthesized attributes of tokens are provided by the scanner; inherited attributes of the
   root are provided by the language definition.

10. Explain the distinction between strong and static type checking.
    Answer: A language is strongly typed if no value can ever be used in a context that expects a value
    of an incompatible type; a language is statically typed if it is strongly typed and the checking can all
    be performed at compile time.

11. State concisely the fundamental tradeoff between static and dynamic type checking.
    Answer: Dynamic type checking provides greater flexibility (e.g. for containers of heterogeneous
    values); static type checking has lower run-time cost and catches errors sooner.

12. 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.

13. For a global variable in C, why is it better to provide an initial value in the declaration, rather
    than in an assignment statement at the beginning of the program?
    Answer: Because the compiler can embed a declared initial value in the object file, and avoid
    the run-time overhead of the assignment statement. (Secondarily, it’s better style—partial credit for
    that.)

14. For a class-typed variable in C++, why is it better to provide an initial value in the declaration, rather
    than in an assignment statement immediately thereafter?
    Answer: Because the second option calls the zero-argument constructor and operator=; the first
    option just calls the single-argument constructor.

15. 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.

16. Why is tail recursion so important in functional languages?
    Answer: Because it can be implemented without allocating a new stack frame.
Multiple Choice

17. What does a compiler need to use to represent the referencing environment of a formal subroutine in C?
   - a. the static link
   - b. the saved contents of the display
   - c. the head pointer of the referencing environment A-list
   - d. nothing at all

18. Why must local variables in Lisp generally be allocated in the heap, rather than on the stack?
   - a. Because local variables in Lisp have unlimited extent.
   - b. Because we don’t know their sizes at compile time.
   - c. Because Lisp subroutines don’t follow strict LIFO calling conventions.
   - d. Because the Lisp garbage collector is unable to manipulate addresses in the stack.

19. 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. binary search in an array
   - d. balanced search tree

20. 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.

21. Why does ML require that all elements of a list have the same type (in contrast to Lisp, 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 ML’s richer syntax (in contrast to Lisp, where everything is an S-expression).
22. What is (are) the benefit(s) of assignment operators (e.g. += in C)?

**Answer:** In addition to being convenient, they calculate the address of their left-hand side only once, avoiding redundant computations and repeated occurrences of side effects.

23. In the 1960s and 1970s, it was generally understood that human beings could write better (faster-running) assembly language than compilers could; the purpose of a compiler was to generate code faster, with fewer mistakes. Around the mid 1980s, however, things began to change. As a general rule, compilers today write better (faster-running) code than people can. What accounts for this change?

**Answer:** Two things: compiler technology has improved, and processor implementations (not ISAs) have become more complex.

24. Consider the following code fragment and its possible translation into assembly language:

```plaintext
int A, B;
...
for i in A..B do
  // whatever
end;

r1 = B
r2 = A
if r2 > r1 goto end
  top:
    // whatever
    r2++
    if r2 <= r1 goto top
end:
```

What is wrong with this translation?

**Answer:** It’s an infinite loop if B == maxint.

25. List six things that a compiler generally allocates in static storage (rather than on the stack or in the heap).

**Answer:** Possibilities include: global variables, static variables, constants, instructions, jump tables for switch statements, symbol table information, address tables for exception handlers, vtables, type descriptors, linkage tables for shared libraries.

26. What is the most significant semantic difference between semaphores and monitor condition variables?

**Answer:** Semaphores remember V operations, but monitor condition variables are memory-less, so excess signal operations are lost.

27. Briefly state the fundamental tradeoff between applicative- and normal-order evaluation of arguments in functional languages.

**Answer:** Applicative-order evaluation is usually faster, at least for arguments that are actually used, but normal-order evaluation is more general: it permits the execution of certain programs that will crash or fail to terminate with applicative-order evaluation.

28. What is the difference between a pointer and an address?

**Answer:** A pointer is an abstraction (a reference-valued object), of which an address is one possible implementation.
Multiple Choice

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.

29. Which of these three operators is type-safe?

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

30. Consider the following C++ code fragment:

```cpp
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?

- X a. `static_cast`
- b. `dynamic_cast`
- c. `reinterpret_cast`
- d. none of the above

31. 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.
- X 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.


32. Consider the following code to initialize an array in C:

```c
double A[100][100];
...
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        A[i][j] = i * j;
    }
}
```

Explain why the version on the left may run significantly faster than the version on the right.

**Answer:** The version on the left walks the array in row-major order, and will experience significantly better spatial locality in the L1 D-cache.

33. Why isn’t it possible to add compacting garbage collection to C or C++?

**Answer:** These languages aren’t fully type-safe, so the garbage collector can’t be sure where all the pointers are.

34. The standard implementation of Python allows only one thread to run at a time, even on a multicore machine. Why, then, must the programmer use locks to protect data structures shared between the main program and event handlers?

**Answer:** Since an event can occur at any time, including when the main program is in the middle of a critical section, we have to make sure that the handler executes only when the main thread has reached a consistent state.

35. How does an in-line subroutine differ semantically from a macro?

**Answer:** It introduces a nested scope and (in most languages) uses applicative-order evaluation for arguments, rather than normal-order.

36. Why do so many languages provide *both* subtype polymorphism and explicit parametric polymorphism (generics)?

**Answer:** The two mechanisms serve different purposes: subtype polymorphism allows the programmer to create a hierarchy of abstractions; generics allow the programmer to instantiate those hierarchies for different constituent types. Put another way: subtype methods can’t easily return values that are usable in a context that expects the supertype; generics don’t support inheritance.

37. Implementations of structured exception handling often create a hidden handler for the body of every function. What purpose(s) does this handler serve?

**Answer:** It performs any necessary cleanup for the function (e.g., calling local variable destructors in C++), deallocates the top-most stack frame, and re-raises the exception at the point of call.

38. Implementations of mix-in inheritance typically embed a special constant at the beginning of the vtable for an interface. What purpose does this constant serve?

**Answer:** It specifies the value that must be subtracted from the interface view of the object (the address of the vtable pointer) to obtain the object pointer to pass as `this` when calling an interface method.
Multiple Choice

39. *Type inference* refers to

- a. invoking the version of a method corresponding to an object’s concrete type, rather than the (abstract) type of the reference used to access it.
- X b. determining the type of an expression based on the types of its constituent parts and/or the surrounding context.
- o c. automatically converting an expression to the type expected by the surrounding context.
- o d. selecting the correct version of an overloaded function based on the types of its arguments.

40. 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 even 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?

```plaintext
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
- X c. 240 bytes
- d. 200 bytes

41. Once upon a time, programs using row-pointer layout for arrays could run faster than programs using contiguous layout, but this is no longer true. Why the change?

- a. Memory accesses used to be faster than multiplication; now it’s the other way around.
- X b. Contiguous layout has significantly better cache locality, which matters a lot on modern machines.
- o c. Compilers have gotten better at pre-calculating the constant parts of contiguous array address calculations.
- o d. Contiguous array address calculations are easier to pipeline, and pipelining is crucial for good performance on modern machines.

42. In which of the following situations is a spin lock an appropriate mechanism for mutual exclusion?

- X a. Synchronization among threads of a scientific simulation, running on a parallel supercomputer.
- o b. Synchronization between a graphical program and its signal handlers.
- o c. Synchronization between processes connected by a Unix pipe (as in `grep foo *.c | less`).
- o d. Control of access to a critical section that writes data to a shared file.
43. (Extra Credit; 6 points max) Generics were initially omitted, deliberately, from Java. They were added in Java 5. Discuss how the resulting design reflects the desire to maintain backward compatibility.

**Answer:** The initial intent in Java was to support polymorphism by falling back on ancestor classes, notably the root class `Object`. When programmers objected, the language designers sought to add generics without changing the virtual machine, duplicating existing (nongeneric) libraries, or breaking existing code. The resulting design is based on the notion of *type erasure*, in which every generic type parameter is effectively treated as `Object`, but the programmer can omit the usual casts on objects returned by library methods. Type parameters were added to all appropriate library classes, but existing code still worked because the parameters were all optional; if missing, they became `Object` by default, which is what all existing code was using. Unfortunately, to maintain compatibility with existing virtual machines (which insisted on type safety), compilers have to generate the type checks omitted by the programmer, so while programs look simpler, they don’t run any faster.

44. (Extra Credit; 6 points max) If C++ had nested subroutines, we might write the code shown here on the left:

```cpp
type def int (*int_func)(int);
void foo(int_func F) {
    cout << F(3) << endl;
}

int main() {
    int N;
    int g(int m) {
        return m + N;
    }
    N = 2;
    foo(&g);
}
```

```cpp
class IF {
public:
    virtual int operator()(int) = 0;
};

void bar(IF *F) {
    cout << (*F)(3) << endl;
}

int main() {
    int main() {
        int N;
        int g(int m) {
            return m + N;
        }
        N = 2;
        foo(&g);
    }
}
```

```cpp
class adder : public IF {
public:
    int _n;
    adder(int n) : _n(n) { }
    virtual int operator()(int m) {
        return m + _n;
    }
};

adder A(2);
bar(&A);
```

In standard C++ (without nested subroutines), we can write the code on the right. Compare these two approaches. Does the ability to write the code at right eliminate the need for nested subroutines?

**Answer:** In the code at right, `adder` member variable `_n` plays roughly the same role as local variable `N` at left. Unlike `N`, however, `_n` is not directly accessible to the code of `main`, and cannot be updated after initialization without making it public or adding a setter method. The designers of C++ would probably argue that object closures (the idiom on the right) suffice to eliminate the need for function closures (the idiom on the left). The designers of C#, Eiffel, Python, and Ruby, on the other hand, would seem to disagree: all allow at least anonymous functions (lambda expressions) to nest, avoiding significant amounts of syntactic clutter in the process, e.g., when setting up event handlers. In the end, this is basically a matter of opinion.