Directions; PLEASE READ

This exam comprises 21 multiple-choice questions, a smaller number of short-answer questions, and one slightly longer extra-credit question. Values are indicated for each; they total 56 points. The extra credit question is worth up to 6 additional points; it isn’t part of the 56, and it won’t factor into your exam score, but it may help to raise your letter grade at the end of the semester.

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, unless otherwise instructed, 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, 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 must complete the exam in class. I will collect any remaining exams promptly at 3:15 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.”

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 (2 points each)

3. Many compilers are divided into two largely independent passes: a front end, responsible for analyzing source code, and a back end, responsible for generating target code. (They may also include a third, intermediate pass, responsible for code improvement.) What is the most important motivation for this division into passes?

   a. to provide multiple combinations of front ends and back ends in a compiler family
   b. to facilitate division of labor when a compiler is built by a large team of programmers
   c. to minimize memory requirements on modern machines
   d. to facilitate debugging the compiler

4. Which of the following mechanisms is intrinsic to context-free languages but not regular languages?

   a. alternation
   b. concatenation
   c. recursion
   d. Kleene closure
5. Which of the following is not true of an LL(1) parser?

○ a. It parses top-down.
○ b. It discovers a left-most derivation.
☒ c. It is also called a shift-reduce parser.
○ d. It parses left-to-right.

Questions 6 through 11 refer to the following LL(1) context-free grammar.

\[
\begin{align*}
\text{dec} & \rightarrow \ \text{dec} \ ; \ \text{dec} _\text{list} \ | \ \epsilon \\
\text{dec} & \rightarrow \ \text{sc} _\text{opt} \ \text{type} \ \text{name} _\text{list} \\
\text{sc} _\text{opt} & \rightarrow \ \text{const} \ | \ \text{static} \ | \ \epsilon \\
\text{type} & \rightarrow \ \text{double} \ | \ \text{int} \\
\text{name} _\text{list} & \rightarrow \ \text{name} \ \text{more} _\text{names} \\
\text{name} & \rightarrow \ \text{id} \ \text{args} _\text{opt} \\
\text{args} _\text{opt} & \rightarrow \ ( \ \text{dec} _\text{list} \ ) \ | \ \epsilon \\
\text{more} _\text{names} & \rightarrow \ , \ \text{name} _\text{list} \ | \ \epsilon
\end{align*}
\]

6. Which of the following strings is not in the language generated by this grammar?

☒ a. int a (int b);
○ b. int c (int d (int e););
○ c. double f, g (static int h;);
○ d. static int i; const double j;

7. How many tokens are in FIRST(\text{dec} _\text{list})?

○ a. 0
○ b. 1
○ c. 2
☒ d. 4

8. How many tokens are in FOLLOW(\text{name})?

○ a. 0
○ b. 1
☒ c. 2
○ d. 4

9. How many tokens are in PREDICT(\text{sc} _\text{opt} \rightarrow \epsilon)?

○ a. 0
○ b. 1
☒ c. 2
○ d. 4
10. Consider a leftmost derivation of the string `static int f();` under this grammar. The first four lines of this derivation are:

```
dec_list
  dec ; dec_list
  sc_opt type name_list ; dec_list
  static type name_list ; dec_list
```

What is the next line?

- a. `static int name_list ; dec_list`
- b. `static type name more_names ; dec_list`
- c. `static type name_list ;`
- d. none of the above

11. Immediately after performing the prediction associated with the answer to the previous question, what symbols would lie on the parse stack of a table-driven LL(1) parser?

- a. `int id ( ) ;`
- b. `static int`
- c. `int name_list ; dec_list`
- d. none of the above

Questions 12 and 13 refer to the following attribute grammar, based on the earlier CFG. This grammar serves to translate a declaration list into a Lisp-like notation, and leave it in the `c` attribute of the root of the parse tree. (Note: you do not have to know Lisp in order to answer these questions.)

```
dec_list → dec ; dec_list
  dec_list → ε
  dec → sc_opt type name_list
    sc_opt → const
    sc_opt → static
    sc_opt → ε
    type → double
    type → int
  name_list → name more_names
  name → id args_opt
  args_opt → ( dec_list )
  args_opt → ε
  more_names → , name_list
  more_names → ε
```

```
▷ dec_list1.c = (cons dec.c dec_list2.c )
▷ dec_list = nil
▷ name_list.s = sc_opt.c
▷ name_list.t = type.c
▷ dec.c = name_list.c
▷ sc_opt.c = 'const
▷ sc_opt.c = 'static
▷ sc_opt.c = nil
▷ type.c = 'double
▷ type.c = 'int
▷ more_names.s = name_list.s
▷ more_names.t = name_list.t
▷ name_list.c = (cons (list (car name.c )
  name_list.s name_list.t (cdr name.c )) more_names.c )
▷ name.c = (cons id.c args_opt.c )
▷ args_opt.c = dec_list.c
▷ args_opt.c = nil
▷ name_list.s = more_names.s
▷ name_list.t = more_names.t
▷ more_names.c = name_list.c
▷ more_names.c = name_list.c
```

3
12. The c, t, and s attributes of name_list are
   ○ a. all synthesized
   ☒ b. synthesized, inherited, and inherited, respectively
   ○ c. inherited, synthesized, and synthesized, respectively
   ○ d. all inherited

13. Which of the following is true?
   ○ a. The attribute grammar is S-attributed.
   ☒ b. The attribute grammar is L-attributed, but not S-attributed.
   ○ c. The attribute grammar is neither S-attributed nor L-attributed.
   ○ d. The attribute grammar is not well formed.

14. Which of the following is not commonly found in a stack frame (activation record)?
   ○ a. return address
   ☒ b. static variables
   ○ c. saved registers
   ○ d. parameters

15. 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

16. 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.

17. Which of the following is considered a pure functional language?
   ○ a. Lisp
   ○ b. ML
   ☒ c. Haskell
   ○ d. Scheme

18. A higher-order function is
   ☒ a. one that takes a function as a parameter, or returns a function as a result.
   ○ b. one at the outermost level of lexical nesting.
   ○ c. one whose local variables have unlimited extent.
   ○ d. one that has been encapsulated with its referencing environment in a closure.
19. Many shell and scripting languages allow strings to be delimited with either single (') or double (") quotes. Why both?

- a. Single-quoted strings are ASCII; double-quoted strings are Unicode.
- b. Single-quoted strings represent a single character; double-quoted strings can be longer.
- c. Variable names are interpolated inside double-quoted strings, but not inside single-quoted strings.
- d. Single-quoted strings are allocated in the stack; double-quoted strings are allocated in the heap.

Questions 20 through 23 are worth one point each.

Suppose that we are programming in Java, that A has been declared as double[] A = new double[10];, and that n is an integer that has the value 15 at run time. Characterize the error in each of the following lines of code.

20. double mole + 6.022e+23; // + should be =

- a. lexical error
- b. syntax error
- c. static semantic error
- d. dynamic semantic error

21. int mole = 6.022e+23; // 6.022e+23 is not an integer

- a. lexical error
- b. syntax error
- c. static semantic error
- d. dynamic semantic error


- a. lexical error
- b. syntax error
- c. static semantic error
- d. dynamic semantic error


- a. lexical error
- b. syntax error
- c. static semantic error
- d. dynamic semantic error
Short Answer

24. (4 points) Describe, in **no more than 15 words**, the set of strings generated by the following regular expression:

\[((a | c)*b (a | c)*b)*(a | c)*\]

**Answer:** Strings of as, bs, and cs with an even number of bs.

25. (a) (4 points) Consider the following function in OCaml:

```ocaml
let f1 l1 l2 =
  let rec g l3 l4 r =
  match l3 with
  | [] -> r
  | h1 :: t1 -> match l4 with
  | [] -> r
  | h2 :: t2 -> match compare h1 h2 with
  | -1 -> g t1 l4 r
  | 0 -> g t1 t2 (h1 :: r)
  | _ -> g l3 t2 r in
  rev (g l1 l2 []);
```

Explain what the function does, in **no more than 15 words**. (Recall that `rev` reverses the order of elements in a list, and that `compare` returns -1, 0, or 1, depending on whether its first argument is less than, equal to, or greater than its second argument.)

**Answer:** Assuming `l1` and `l2` are sorted, it finds the elements present in both.

(b) (4 points) Consider the following alternative version (differences underlined):

```ocaml
let f1 l1 l2 =
  let rec g l3 l4 r =
  match l3 with
  | [] -> r
  | h1 :: t1 -> match l4 with
  | [] -> r
  | h2 :: t2 -> match compare h1 h2 with
  | -1 -> g t1 l4 r
  | 0 -> g t1 t2 (r @ [h1])
  | _ -> g l3 t2 r in
  g l1 l2 [];;
```

Explain why the first version is probably better.

**Answer:** Each invocation of the `append` (@) function is likely to need to scan to the end of the current list. Total execution time will then be O(n^2), where n is the length of the input. By building the list in reverse order (at constant time per element) and then reversing it at the end, we get O(n) time instead.

26. (3 points) What is (are) the initial source(s) of information in the decoration of (evaluation of attributes in) a parse or syntax tree?

**Answer:** Synthesized attributes of tokens are provided by the scanner; inherited attributes of the root reflect the language definition or execution environment.
Extra Credit

27. (6 points max) Lambda expressions were recently added to C++. The `add_n` function discussed in class could be written

```cpp
std::function<int(int)> add_n(int n) { return [n](int m){ return n + m;} ; }
```

The lambda expression here has been underlined. The `n` in square brackets indicates that we should capture the (current) value of `n`. Alternatively, we could have put `&n` inside the brackets, indicating that we should capture `n`’s *address* instead, and allow the lambda to access it indirectly. Discuss this notion of explicit capture. Why is it needed? What are the tradeoffs between capturing values and capturing addresses?

**Answer:** We need explicit capture because C++ has no garbage collection, and cannot easily arrange for variables used from within lambdas to have unlimited extent. If a variable is changed after we capture its value, then the lambda, when called, will not see the change. If we capture an address, then changes will be visible, but we run the risk of following a dangling reference if the variable’s lifetime ends. In the example above, capture by value is safe; capture by address is not.