Directions—PLEASE READ

This exam comprises a mix of multiple-choice and short-answer questions, plus one slightly longer extra-credit problem. Values are indicated for each. The regular questions total 58 points (including 2 points for putting your name on every page). The extra credit question is worth up to 8 additional points; it isn’t part of the 58, 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 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, the proctor 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. Any remaining exams will be collected promptly at 11:40 am. 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. (2 points) Put your name on every page (so if I lose a staple I won’t lose your answers).
Multiple Choice (2 points each)

3. Which of the following languages is the newest?

○ a. C  
○ b. Fortran  
○ c. Lisp  
○ d. Java

4. Which of the following languages is the oldest?

○ a. Perl  
○ b. PHP  
○ c. Python  
○ d. Ruby

5. Which of the following is typically not the job of the scanner?

○ a. remove comments  
○ b. expand macros  
○ c. tag tokens with line numbers  
○ d. save text of identifiers, numbers, strings, etc.

6. What is the main reason why a language designer might choose to make the scope of a name run from its declaration to the end of the block (rather than covering the entire block)?

○ a. It makes intuitive sense in a language that requires all names to be declared before they are used.  
○ b. It makes programs easier to compile, because the parser never has to read ahead to find a declaration.  
○ c. It facilitates separate compilation.  
○ d. It avoid the logical paradox of circular definitions.

7. What is the main reason why a language designer might choose to make the scope of a name be the entire block in which it is declared (rather than just the portion from the declaration to the end of the block)?

○ a. It allows compiled programs to run faster, since all names in a scope can be introduced at once, rather than incrementally.  
○ b. It avoids any ambiguity about the referencing environment to be captured in a closure.  
○ c. It facilitates separate compilation.  
○ d. It makes it easy to write mutually recursive definitions.
8. What is the principal motivation for syntax error recovery?
   - a. to allow the compiler to output better error messages
   - b. to figure out what the programmer meant to say, so we can proceed with code generation
   - c. to allow the compiler to keep looking for additional errors
   - d. none of the above

9. The static chain
   - a. allows a running program to find variables whose lifetime spans the entire program execution.
   - b. identifies objects imported into the current scope.
   - c. allows a subroutine to find the frame of the lexically surrounding routine.
   - d. identifies all modules that were linked together in a single object file.

10. A higher-order function is
   - a. one at the outermost level of lexical nesting
   - b. one that takes other functions as parameters, or returns a function as a result
   - c. one whose local variables have unlimited extent
   - d. one that has been encapsulated with its referencing environment in a closure

11. Gimple is
   - a. a scripting language intended for video games
   - b. a popular implementation of the Scheme programming language
   - c. an industry-standard representation for symbol-table information
   - d. the AST representation used by the gcc compilers

Short Answer (values as marked)

12. (4 points). Would it be reasonable to demand a tail-recursive version of the classic quicksort algorithm? Why or why not?
13. (4 points). Describe in simple English the difference between the representations of decimal numbers defined by the following two extended regular expressions. (Do not discuss low-level details of the regular expressions themselves.) Note that the parentheses, plus, star, and vertical bar are all meta-symbols; the dots and digits are actual characters.

\[ \text{digit}^+ . \text{digit}^+ \]
\[ ( \text{digit}^* . \text{digit}^+ ) \mid ( \text{digit}^+ . \text{digit}^* ) \]

14. (5 points). Show the minimal DFA corresponding to the second regular expression in the preceding problem. Hint: you can probably solve this by inspection, without running the formal construction.
Questions ?? through ?? (2 points each) refer to the following function in OCaml:

```ocaml
let x = 1;;
let f () =
    let x = 2 in
    let c p = let x = 4 in p () in
    let d () = x in
    let b () = let x = 3 in c d in
    b ();;
```

Note that f, d, and b are all zero-argument functions.

15. What will function f return when called?

   ○ 1   ○ 2   ○ 3   ○ 4

16. What would f return if OCaml had dynamic scope and shallow binding?

   ○ 1   ○ 2   ○ 3   ○ 4

17. What would f return if OCaml had dynamic scope and deep binding?

   ○ 1   ○ 2   ○ 3   ○ 4

In Questions ?? through ?? (1 point each), indicate whether the specified program error in C++ is (in most cases) a lexical error, syntax error, static semantic error, dynamic semantic error, or none of the above (may go undetected, resulting in undefined behavior).

18. expression with unbalanced parentheses

19. ‘Ø’ character not in a comment or string/character constant

20. array subscript out of bounds

21. uncaught exception

22. call to function with wrong number of arguments
Questions ?? through ?? refer to the following LL(1) context-free grammar:

\[
\begin{align*}
decs & \rightarrow \text{VAR} \ dec \ more\_decs \ $$$ \\
dec & \rightarrow \ text{ID} \ ; \\
more\_decs & \rightarrow \ dec \ more\_decs \\
& \rightarrow \\
id\_list & \rightarrow \ ID \ more\_ids \\
more\_ids & \rightarrow \ , \ id\_list \\
& \rightarrow \\
\end{align*}
\]

The ID after the colon in the second production is meant to be a type name.

**Multiple Choice** (2 points each)

23. What is FIRST(\text{more\_decs})?

- a. \{ID\}
- b. \{ID, ;\}
- c. \{$$\}
- d. \{$$, ;\}

24. What is FOLLOW(\text{dec})?

- a. \{ID\}
- b. \{ID, ;\}
- c. \{$$\}
- d. \{ID, $$\}

25. What is PREDICT(\text{more\_ids → ϵ})?

- a. \{;\}
- b. \{;\}
- c. \{$$\}
- d. \{ϵ\}
26. Consider the string \texttt{VAR A, B : FOO ; C : BAR ; \$$. The first four lines of a left-most
derivation of this string are

\begin{verbatim}
  decs
  VAR dec more_decs $$
  VAR id_list : ID ; more_decs $$
  VAR ID more_ids : ID ; more_decs $$
\end{verbatim}

What is the next line?

- a. \texttt{VAR ID more_ids : ID ; dec more_decs $$}
- b. \texttt{VAR ID, ID more_ids : ID ; more_decs $$}
- c. \texttt{VAR ID, id_list : ID ; more_decs $$}
- d. \texttt{VAR ID, id_list : ID ; dec more_decs $$}

27. How long is the entire derivation?

- a. 9 lines
- b. 10 lines
- c. 11 lines
- d. 12 lines

28. Consider a recursive descent parse of this same token string. At the point where the identifier
\texttt{C} is matched, which recursive descent routines will be active (i.e., which frames, from bottom
to top, will be on the parser's run-time stack)?

- a. \texttt{decs dec id_list}
- b. \texttt{decs more_decs dec}
- c. \texttt{decs more_decs dec id_list}
- d. \texttt{decs more_decs dec id_list more_ids}

29. At that same exact point in the execution of a table-drive top-down parser, what symbols
will be in the (explicitly managed) parse stack?

- a. \texttt{: ID ; $$}
- b. \texttt{more_ids more_decs}
- c. \texttt{decs more_decs dec id_list}
- d. \texttt{more_ids : ID ; more_decs $$}
30. (Extra Credit; up to 8 points) Write an attribute grammar for our declaration language that will insert each declared variable into the symbol table, together with its type. You should assume that the symbol table is a global structure, not something passed through attributes, and that variables are added to it by calling `declare_variable(string var_name, string type_name)`. You may assume that each identifier’s character-string name is less than some language-defined maximum in length, and is available in a synthesized attribute `n` of type `string` attached to the `ID` token by the scanner. You may invent any other attributes you need to support your action routines, but to make the task interesting, each attribute must be of bounded size (big enough to hold a variable or type name, but not a list of variable names or a string of unbounded length). Hint: your attribute grammar won’t be complicated, but it won’t be L-attributed either.

\[
\text{decs} \rightarrow \text{VAR} \text{ dec more_decs} \$$ \\
\text{dec} \rightarrow \text{id_list} : \text{ID} ; \\
\text{more_decs} \rightarrow \text{dec more_decs} \\
\quad \rightarrow \\
\text{id_list} \rightarrow \text{ID more_ids} \\
\text{more_ids} \rightarrow , \text{id_list} \\
\quad \rightarrow 
\]