Midterm Exam

CSC 2/454

21 October 2015

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.”

   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 (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 ?? through ?? refer to the following LL(1) context-free grammar.

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

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 \textsc{first}(dec\_list)\
\begin{itemize}
\item [\textbullet] a. 0
\item [\textbullet] b. 1
\item [\textbullet] c. 2
\item [\textbullet] d. 4
\end{itemize}

8. How many tokens are in \textsc{follow}(name)\
\begin{itemize}
\item [\textbullet] a. 0
\item [\textbullet] b. 1
\item [\textbullet] c. 2
\item [\textbullet] d. 4
\end{itemize}

9. How many tokens are in \textsc{predict}(\textit{sc\_opt} \rightarrow \epsilon)\
\begin{itemize}
\item [\textbullet] a. 0
\item [\textbullet] b. 1
\item [\textbullet] c. 2
\item [\textbullet] d. 4
\end{itemize}

10. Consider a leftmost derivation of the string \texttt{static int f();} under this grammar. The first four lines of this derivation are:
\begin{align*}
\textsc{dec\_list} \\
\texttt{dec ; dec\_list} \\
\textit{sc\_opt type name\_list ; dec\_list} \\
\texttt{static type name\_list ; dec\_list}
\end{align*}
What is the next line?
\begin{itemize}
\item [\textbullet] a. \texttt{static int name\_list ; dec\_list}
\item [\textbullet] b. \texttt{static type name more\_names ; dec\_list}
\item [\textbullet] c. \texttt{static type name\_list ;}
\item [\textbullet] d. none of the above
\end{itemize}

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?
\begin{itemize}
\item [\textbullet] a. \texttt{int id ( ) ;}
\item [\textbullet] b. \texttt{static int}
\item [\textbullet] c. \texttt{int name\_list ; dec\_list}
\item [\textbullet] d. none of the above
\end{itemize}
Questions ?? and ?? 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.)

\[
\begin{align*}
\text{dec_list} & \rightarrow \text{dec} ; \text{dec_list} & \rightarrow \text{dec} & \rightarrow \epsilon \\
\text{dec} & \rightarrow \text{sc_opt type name_list} & \text{sc_opt} & \rightarrow \text{const} & \text{sc_opt} & \rightarrow \text{static} & \text{sc_opt} & \rightarrow \epsilon \\
\text{name_list} & \rightarrow \text{name more_names} & \text{type} & \rightarrow \text{double} & \text{type} & \rightarrow \text{int} \\
\text{name} & \rightarrow \text{id args_opt} & \text{args_opt} & \rightarrow (\text{dec_list}) & \text{args_opt} & \rightarrow \epsilon & \text{more_names} & \rightarrow , \text{name_list} \\
\text{more_names} & \rightarrow \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{dec_list}_1.c &= (\text{cons dec.c dec_list}_2.c) \\
\text{dec_list} &= \text{nil} \\
\text{name_list}.s &= \text{sc_opt.c} \\
\text{name_list}.t &= \text{type.c} \\
\text{dec.c} &= \text{name_list.c} \\
\text{sc_opt.c} &= '\text{const}' \\
\text{sc_opt.c} &= '\text{static}' \\
\text{sc_opt.c} &= \text{nil} \\
\text{type.c} &= '\text{double}' \\
\text{type.c} &= '\text{int}' \\
\text{more_names}.s &= \text{name_list.s} \\
\text{more_names.t} &= \text{name_list.t} \\
\text{name_list.c} &= (\text{cons (list (car name.c) name_list.s name_list.t (cdr name.c)) more_names.c}) \\
\text{name.c} &= (\text{cons id.c args_opt.c}) \\
\text{args_opt.c} &= \text{dec_list.c} \\
\text{args_opt.c} &= \text{nil} \\
\text{name_list.s} &= \text{more_names.s} \\
\text{name_list.t} &= \text{more_names.t} \\
\text{more_names.c} &= \text{name_list.c} \\
\text{more_names.c} &= \text{nil}
\end{align*}
\]

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 ?? through ?? 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:

\[( (a | c)^* b (a | c)^* b)^* (a | c)^* \]
25. (a) (4 points) Consider the following function in OCaml:

```ocaml
let f 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.)

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

```ocaml
let f 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.
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?

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?