Midterm Exam
CSC 254
24 October 2016

Directions—PLEASE READ

This exam comprises a mix of multiple-choice and short-answer questions, together with one slightly longer extra-credit question. Values are indicated for each. The regular questions total 68 points. The extra credit question is worth up to 8 additional points; it isn’t part of the 68, 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, 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. 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.”

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. What is the oldest high-level (non-assembler) programming language?
   - a. Basic
   - b. C
   - c. Fortran
   - d. Lisp

4. 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 the classic Unix yacc tool.
5. What is the difference between synthesized and inherited attributes?

- a. Synthesized attributes are initialized by the scanner.
- b. Synthesized attributes depend only on information below them in the parse tree.
- c. Inherited attributes are useful only in object-oriented languages.
- d. Synthesized attributes are computed at run time.

6. Closures, which combine a subroutine pointer and a referencing environment,

- a. are needed only for languages with static (not dynamic) scope.
- b. are needed only for first-class (not second- or third-class) subroutines.
- c. are needed only for deep (not shallow) binding.
- d. All of the above.

Questions 7 through 12 refer to the following program in OCaml:

```ocaml
let add_n n m = n + m;;
let n = 4 in
let f = add_n 3 in
n + (f 2);;
```

7. What does this program print?

- a. 8
- b. 9
- c. 10
- d. none of the above

8. What would the program print if OCaml used dynamic scope and shallow binding?

- a. 8
- b. 9
- c. 10
- d. none of the above

9. What would the program print if OCaml used dynamic scope and deep binding?

- a. 8
- b. 9
- c. 10
- d. none of the above
10. The fact that the program contains two variables named \( n \) is an example of

- a. overloading
- b. aliasing
- c. both
- d. neither

11. The fact that \texttt{add\_n} is passed a single argument at line 3 is an example of

- a. dynamic binding
- b. currying
- c. unlimited extent
- d. none of the above

12. What would happen (in real OCaml) if we reversed the second and third \texttt{let} clauses (the definitions of \( n \) and \( f \))?

- a. Nothing: the program would behave the same.
- b. The output would change, because \( f \) would capture a different \( n \).
- c. The interpreter would complain that \( n \) is being used before it is declared.
- d. The interpreter would complain that the meaning of \( n \) is ambiguous.

\textbf{Short Answer}

13. (3 points) Why is tail recursion so important in functional languages?

\textbf{Answer:} Because it can be implemented without allocating a new stack frame.

14. (3 points) Explain why the lists are generally homogeneous (all elements the same type) in OCaml, but often heterogeneous (elements of varying types) in Scheme.

\textbf{Answer:} OCaml is statically typed; Scheme is dynamically typed. Without homogeneity (or a solution to the halting problem :-) there is no straightforward way to ensure at compile time that no element of a list will ever be used in a way that is inconsistent with its type.

15. (3 points) Explain the difference between the \textit{static link} and the \textit{dynamic link} in a stack frame.

\textbf{Answer:} The dynamic link is the saved frame pointer; it points to the frame of the calling subroutine. In a language with lexical scope and nested subroutines, the static link points to the frame of the lexically surrounding subroutine.

16. (5 points) List, in order, the principal phases of a typical compiler. Which of these may also be found in an interpreter?

\textbf{Answer:} Scanning, parsing, semantic analysis, intermediate code generation, machine-independent code improvement, target code generation, machine-specific code improvement. The first three are present in all interpreters. The fourth will also almost certainly be present, especially if we think of an abstract syntax tree as intermediate code. A high-quality interpreter may also include some form of machine-independent code improvement.
17. (4 points) Describe, in no more than 20 English words, the language (set of strings) accepted by the following nondeterministic finite automaton. (I’m looking for an intuitive description of the language, not a description of what the machine does.)

Answer: All strings of as, bs, and cs with at least three of the same letter in a row.

18. (4 points) How many states are there in the minimal equivalent deterministic finite automaton? (Note: you should be able to figure this out without working through the construction.)

Answer: 8: a start state, a final state, and one state each to indicate that the machine has just seen one a, two as, one b, two bs, one c, or two cs in a row.

19. (4 points) Name at least two features of the “advanced regular expressions” of Perl and other scripting languages that make them strictly more powerful than classical regular expressions (as used, for example, in scanner generators). (Note: I’m looking here for features that allow you to describe sets of strings that you could not describe with classical REs—not for abbreviations that simply make it easier to describe things you could describe before.)

Answer: Possibilities include variable capture, minimal matches, backreferences, variable interpolation, and embedded code.

Questions 20 through 24 and the extra-credit question (number 25) use the following LL(1) grammar (this is essentially a simplified subset of Lisp, but you don’t have to know anything about Lisp to answer these questions):

\[
\begin{align*}
P & \to E \\
E & \to \text{atom} \\
& \to ' E \\
& \to ( E E s ) \\
E s & \to E E s \\
& \to \epsilon
\end{align*}
\]

// this is an epsilon production

20. (6 points)

(a) What is FIRST(Es)?

Answer: \{ \text{atom, (, } \}

(b) What is FOLLOW(E)?

Answer: \{ \text{atom, (, ), }\}, \$$\}

(c) What is PREDICT(Es \to \epsilon)?

Answer: \{ \}
21. (4 points) Which of the following is a correct parse tree for the string \((\text{cdr } '(\text{a b c}))\), where \text{cdr}, \text{a}, \text{b}, \text{and} \text{c} are all atoms?

- **a.**
  ![Parse Tree a](image1)

- **b.**
  ![Parse Tree b](image2)

- **c.**
  ![Parse Tree c](image3)

- **d.**
  ![Parse Tree d](image4)

22. (4 points) What is the sixth line of a leftmost derivation of \((\text{cdr } '(\text{a b c}))\)?

- **X a.** \((\text{cdr } ' (\text{E Es}))\) $$
- **b.** \((\text{E } ' \text{E})\) $$
- **c.** \(\text{E Es})\) $$
- **d.** \(\text{E Es})\) $$
- **d.** \(\text{E } ' \text{E})\) $$
23. (3 points) Consider a recursive descent parse of the string \( \text{cdr } '(a b c) \). At the point where the quote token ('') is matched, which recursive descent routines will be active (i.e., what frames [in what order] will be on the parser’s run-time stack)?

**Answer:** P, E, Es, E, and match.

24. (3 points) At that exact same point in the execution of a table-driven top-down parser, what symbols will be in the (explicitly managed) parse stack?

**Answer:** ' E Es ) $$

25. (8 points **Extra Credit only**) One of the interesting characteristics of Lisp and its descendants is that programs are represented as lists. Since each list is actually a (head, rest) pair—what Lisp calls a cons cell—the natural syntax tree for a Lisp program is a tree of such cells. The syntax tree for \( \text{cdr } '(a b c) \) is shown at right. (Note that 'L is syntactic sugar for \( \text{quote L} \).)

Extend the CFG on page 4 to create an attribute grammar that will build such trees. When a parse tree has been fully decorated, the root should have an attribute \( v \) that refers to the syntax tree. You may assume that each atom has a synthesized attribute \( v \) that refers to a syntax tree node that holds information from the scanner. In your semantic functions, you may assume the availability of a cons function that takes two references as arguments and returns a reference to a new cons cell containing those references.

**Answer:**

\[
\begin{align*}
P & \rightarrow E \quad \triangleright \quad P.v = E.v \\
E & \rightarrow \text{atom} \quad \triangleright \quad E.v = \text{atom}.v \\
E_1 & \rightarrow ' E_2 \quad \triangleright \quad E_1.v = \text{cons}('v, \text{cons}(E_2.v, \text{nil})) \\
E_1 & \rightarrow ( E_2 Es ) \quad \triangleright \quad E_1.v = \text{cons}(E_2.v, Es.v) \\
Es_1 & \rightarrow E Es_2 \quad \triangleright \quad E_1.v = \text{cons}(E.v, E_2.v) \\
Es & \rightarrow \epsilon \quad \triangleright \quad Es.v = \text{nil}
\end{align*}
\]