Midterm Exam
CSC 254
20 October 2011

Directions—PLEASE READ

This exam comprises a mixture of multiple-choice, short-essay and problem-solving questions. Values are indicated for each; they total 60 points. Question 21 is worth 6 extra credit points; it isn’t part of the 60, and won’t factor into your exam score, but 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 has been instructed not 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 1:45 pm. Good luck!

1. (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)

2. What is the oldest high-level (non-assembler) programming language?

   ○ a. C.
   ○ b. Lisp
   × c. Fortran
   ○ d. Basic

3. 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.

4. 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.
Questions 5 through 9 refer to the following program in Scheme:

```scheme
(define add-n (lambda (n) (lambda (m) (+ m n))))
(let ((n 4)
      (f (add-n 3)))
  (+ n (f 2)))
```

5. What does this program print?
   - a. 8
   - b. 9
   - c. 10
   - d. none of the above

6. What would the program print if Scheme used dynamic scope and shallow binding?
   - a. 8
   - b. 9
   - c. 10
   - d. none of the above

7. What would the program print if Scheme used dynamic scope and deep binding?
   - a. 8
   - b. 9
   - c. 10
   - d. none of the above

8. The fact that the program contains two variables named n is an example of
   - a. overloading
   - b. aliasing
   - c. both
   - d. neither

9. What would happen (in real Scheme) if we reversed the order of the two definitions in the let block?
   - x. 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.
10. **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.

11. Which of the following is not a higher-order function?

- a. map
- **b. member**
- c. apply
- d. compose

**Short Answer**

12. (3 points) Why is tail recursion so important in functional languages?

**Answer:** Because it can be implemented without allocating a new stack frame.

13. (3 points) Explain the difference between the *static link* and the *dynamic link* in a stack frame.

**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.

14. (3 points) How should one choose whether to implement dynamic web page content on the server side (e.g., in PHP) or the client side (e.g., in Javascript)?

**Answer:** As a general rule, use client-side scripting for low latency. Use server-side scripting for access to resources (e.g., database content) that should not be accessible to the client.

15. (5 points) List, in order, the principal phases of a typical compiler. Which of these may also be found in an interpreter?

**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.

---

**Questions 16 (following page) through 19 and the extra-credit question (number 21) use the following LL(1) grammar for a simplified subset of Scheme:**

\[
\begin{align*}
P & \rightarrow E \$$ \\
E & \rightarrow \text{atom} \\
& \rightarrow 'E \\
& \rightarrow ( E Es ) \\
Es & \rightarrow E Es \\
& \rightarrow // this is an epsilon production
\end{align*}
\]
16. (6 points)

(a) What is FIRST(Es)?
   \textbf{Answer:} \{ \text{atom, (, '} \}

(b) What is FOLLOW(E)?
   \textbf{Answer:} \{ \text{atom, (, ', ), $\$$} \}

(c) What is PREDICT(Es \rightarrow \epsilon)?
   \textbf{Answer:} \{ \} \}

(Note: you are not required to hand-execute the formal algorithm, or show your work. With a grammar this simple you should probably be able to get the right answer just by staring at it for a minute or two.)

17. (5 points) Give a parse tree for the string \textit{(cdr ' (a b c))}, where \textit{cdr}, \textit{a}, \textit{b}, and \textit{c} are all \textit{atoms}. Don't forget that the parentheses are tokens.

\textbf{Answer:}

\begin{center}
\begin{tikzpicture}
  \node (P) at (0,0) {$\text{P}$};
  \node (E) at (-1,1) {$\text{E}$};
  \node (Es) at (0,1) {$\text{Es}$};
  \node (cdr) at (-2,2) {$\text{cdr}$};
  \node (E) at (-1,3) {$\text{E}$};
  \node (Es) at (0,3) {$\text{Es}$};
  \node (E) at (-1,4) {$\text{E}$};
  \node (Es) at (0,4) {$\text{Es}$};
  \node (E) at (-1,5) {$\text{E}$};
  \node (Es) at (0,5) {$\text{Es}$};
  \node (E) at (-1,6) {$\text{E}$};
  \node (Es) at (0,6) {$\text{Es}$};
  \node (E) at (-1,7) {$\text{E}$};
  \node (Es) at (0,7) {$\text{Es}$};
  \node (E) at (-1,8) {$\text{E}$};
  \node (Es) at (0,8) {$\text{Es}$};
  \node (E) at (-1,9) {$\text{E}$};
  \node (Es) at (0,9) {$\text{Es}$};
  \node (E) at (-1,10) {$\text{E}$};
  \node (Es) at (0,10) {$\text{Es}$};
  \node (E) at (-1,11) {$\text{E}$};
  \node (Es) at (0,11) {$\text{Es}$};
  \node (E) at (-1,12) {$\text{E}$};
  \node (Es) at (0,12) {$\text{Es}$};
  \node (E) at (-1,13) {$\text{E}$};
  \node (Es) at (0,13) {$\text{Es}$};
  \node (E) at (-1,14) {$\text{E}$};
  \node (Es) at (0,14) {$\text{Es}$};
  \node (E) at (-1,15) {$\text{E}$};
  \node (Es) at (0,15) {$\text{Es}$};

  \draw (P) -- (E);
  \draw (E) -- (Es);
  \draw (cdr) -- (E);
  \draw (E) -- (Es);
  \draw (E) -- (Es);
  \draw (E) -- (Es);
  \draw (E) -- (Es);
  \draw (E) -- (Es);
  \draw (E) -- (Es);

  \end{tikzpicture}
\end{center}

18. (5 points) Show the first ten lines of the leftmost derivation of \textit{(cdr ' (a b c))}.
(The whole derivation is fourteen lines long.)

\textbf{Answer:} Here is the whole derivation:

\begin{verbatim}
P
E $\$$
  ( E Es ) $\$$
  ( cdr Es ) $\$$
  ( cdr E Es ) $\$$
  ( cdr ' E Es ) $\$$
  ( cdr ' ( E Es ) Es ) $\$$
  ( cdr ' ( a Es ) Es ) $\$$
  ( cdr ' ( a E Es ) Es ) $\$$
  ( cdr ' ( a b Es ) Es ) $\$$
\end{verbatim}
19. (3 points) Consider a recursive descent parse of the expression (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.

20. (5 points) Real number constants in Scheme are permitted to be explicitly *inexact*. A programmer who wants to express all constants using the same number of digits can use the sharp sign (#) as a placeholder for any trailing digit whose value is not known. For example, the following are all approximations (with varying degrees of inexactness) for the constant value 100π:

- 314.16: five digits specified, all known
- 31#: three digits specified, one unknown, no decimal point
- 3##.: three digits specified, two unknown, with decimal point
- 3###.: five digits specified, four unknown
- 314.: four digits specified, one unknown
- 314.2##: six digits specified, two unknown

Similarly, approximations of π/10 could be written as

- 00.314: five digits specified (including two leading zeros), all known
- 0.3##: four digits specified, two unknown
- 0.##: three digits specified, two unknown
- .31###: five digits specified, three unknown, with nothing to left of decimal point

Note that there must always be at least one digit whose value is known; they can’t all be sharp signs.

Give a regular expression for (possibly inexact) real number constants in Scheme, as described informally above. You may use the name `digit` as shorthand for (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9). You may also use the “Kleene plus” shorthand notation if you want.

(By the way, numbers in Scheme are actually a good bit more complicated than this. Please ignore anything you may know about sign, exponent, radix, exactness and length specifiers, and complex or rational values.)

Answer: There is, of course, an infinite number of correct answers. Here is the simplest one I was able to come up with:

\[ ( \text{digit}^+ \text{#}* ( . | \epsilon ) \text{#}* | \text{digit}^* . \text{digit}^+ \text{#}* ) \]
21. (6 points **Extra Credit only**) Because programs are lists in Scheme, the natural syntax tree for a Scheme program is a tree of cons cells. The syntax tree for \((\text{cdr } \text{quote } \text{a b c})\) is shown at right. (Note that \(\text{quote } \text{L}\) is syntactic sugar for \((\text{quote } \text{L})\).)

Extend the CFG on page 3 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:**

\[
\begin{align*}
P & \rightarrow E \quad P.v = E.v \\
E & \rightarrow \text{atom} \quad E.v = \text{atom}.v \\
E_1 & \rightarrow \text{quote } E_2 \quad E_1.v = \text{cons}(:, E_2.v) \\
E_1 & \rightarrow ( E_2 E_s ) \quad E_1.v = \text{cons}(E_2.v, E_s.v) \\
E_s & \rightarrow E \quad E_s.v = \text{cons}(E.v, E_2.v) \\
E_s & \rightarrow \epsilon \quad E_s.v = \text{nil}
\end{align*}
\]