Questions 2 through 8 refer to the following attribute grammar.
(Questions 2 through 6 consider only the context-free part.)

\[
\begin{align*}
P & \rightarrow \ E \ \mathbf{\$\$} & \triangleright & \ E.d = 0 & \triangleright & \ P.m = E.m \\
E & \rightarrow \ A & \triangleright & \ A.d = E.d & \triangleright & \ E.m = A.m \\
E & \rightarrow \ B & \triangleright & \ B.d = E.d & \triangleright & \ E.m = B.m \\
E & \rightarrow \ \epsilon & \triangleright & \ E.m = E.d \\
A & \rightarrow \ ( \ E_1 \ ) \ E_2 & \triangleright & \ E_1.d = A.d + 1 & \triangleright & \ E_2.d = A.d & \triangleright & \ A.m = \max(E_1.m, E_2.m) \\
B & \rightarrow \ [ \ E_1 \ ] \ E_2 & \triangleright & \ E_1.d = B.d + 1 & \triangleright & \ E_2.d = B.d & \triangleright & \ B.m = \max(E_1.m, E_2.m)
\end{align*}
\]

2. (3 points) Describe, in English, the language generated by the context-free grammar. (Give me a high-level description. Do not explain what the various productions do.)

**Answer:** The language consists of all strings of balanced parentheses and brackets.

3. (5 points) List the grammar’s terminals and non-terminals. Identify the start symbol.

**Answer:** Terminals: \{ (, ), [ , ], $$ \} \\
Non-terminals: \{ P, E, A, B \}
Start symbol: \( P \)

4. (8 points) Give the predict set for each production. (You should be able to do this by inspection. You should not have to run through the whole construction from assignment 2.) Is the grammar LL(1)? How do you know?
Answer:
PREDICT\( (P \rightarrow E \)\$: \{ , [ , $$ \}

\text{PREDICT(}E \rightarrow A\): \{ \}\n
\text{PREDICT(}E \rightarrow B\): \{ \}\n
\text{PREDICT(}E \rightarrow \epsilon\): \{ , ] , $$ \}\n
\text{PREDICT(}A \rightarrow (E) E\): \{ \}\n
\text{PREDICT(}B \rightarrow [E] E\): \{ \}\n
\(E\) is the only nonterminal on the left-hand side of more than one production, and the predict sets of those productions are disjoint, so yes, the grammar is LL(1).

5. (6 points) Give a derivation for the string \([()()]\) $$ . (Hint: the answer is 12 lines long.)

Answer:

\(P\)

\(E\) $$

\(B\) $$

\([E] E\) $$

\([A] E\) $$

\([(E) E] E\) $$

\([(E)] E\) $$

\([(E)] E\) $$

\([(E)] E\) $$

\([(E)] E\) $$

\([(E)] E\) $$

\([(E)] E\) $$

\([()] E\) $$

\([()] E\) $$

\([()] E\) $$

\([()] E\) $$

\([()] E\) $$

\([()] E\) $$

\([()] E\) $$

\(6. (6 points) Draw the parse tree for this same string.

Answer:

\(7. (8 points) Now consider the full attribute grammar. List its attributes, and describe in English what each represents. Which attributes are synthesized? Which are inherited? Is the grammar S-attributed? Is it L-attributed?

Answer: \(A.m, B.m, \) and \(P.m\) are synthesized. \(A.d, B.d, E.d, \) and \(E.m\) are inherited. \((E.m\) is tricky. It depends, in one instance, on \(E.d, \) which comes from above.) In each case, attribute \(d\) represents the number of levels of parentheses that surround a symbol in the input string.

2
Attribute \( m \) represents the maximum nesting depth below the symbol in the parse tree. The grammar is not S-attributed, but it is L-attributed.

8. (8 points) Now, using the full attribute grammar, decorate your tree. For \( E \) symbols, indicate \( d \) first, then \( m \).

   **Answer:** See above.

9. (4 points) We often describe compilation and interpretation as alternatives to one another, but Java is sometimes compiled and then interpreted. Explain how this makes sense.

   **Answer:** Java source code is compiled to portable, machine-independent byte code. All static errors are discovered during this initial compilation step. The byte code is much easier than source code to interpret. (It may also be JIT-compiled to machine code.)

10. (5 points) How does a compiler usually handle lexical errors (errors encountered by the scanner)? Why are these easier to handle than the syntax errors discovered by the parser?

    **Answer:** The scanner typically prints an error message and then ignores malformed tokens, effectively deleting them. It doesn’t care about larger structures, and the parser already knows how to recover from missing tokens. When the parser finds an error, however, it can’t just delete the larger structure.

11. (4 points) Why is it that only one language is widely used for client-side web scripting, but many are used for server-side scripting?

    **Answer:** Server-side scripts are executed at a single site, and can use whatever language the site administrator prefers. Client-side scripts must generally be able to run in any browser, on any machine in the world. (Note that while users routinely add plug-ins to their browser, these are much less problematic than a JavaScript alternative would be, because they don’t have access to the DOM, and therefore aren’t as likely to open security holes.)

12. (5 points) Early versions of Fortran didn’t allow recursion, so there would never be more than one active instance of a subroutine, and compilers could use static allocation for local variables, parameters, and bookkeeping information. Many Fortran compilers, however, used a subroutine call stack anyway. Speculate as to why.

    **Answer:** (1) Since it is unlikely that all subroutines will be active at the same time (this would require that code call into all of them before returning from any), a stack will generally take less total space than static allocation. (2) If we want to be able to call libraries written in different languages (some of which may have recursion), we need to use the same calling conventions in all compilers. (I also gave credit for improving locality. This didn’t matter much in the days of early Fortran, but it’s crucial for modern caches.)

13. Consider the following function in Scheme.

\[
\text{(define P1}
\text{  (lambda (n)}
\text{    (lambda (x)}
\text{      (if (= n 0) 1 (* x ((P1 (- n 1)) x))))))
\]

(a) (6 points) What does this function do? For example, what is \((P1 2)\)?

   **Answer:**

What is \(((P1 2) 3)\)?
Answer: P1 generates exponentiation functions (assuming \( n \) is nonnegative). \((P1\ 2)\) returns a function that squares its argument. \((P1\ 3)\) returns a function that cubes it. \(((P1\ 2)\ 3)\) is 9.

(b) (6 points) Explain why it matters that the function returned by P1 uses deep binding.

Answer: \((P1\ 2)\) and \((P1\ 3)\) are both references to the anonymous function inside P1, but with different referencing environments, containing different instances of \( n \). In one environment, \( n \) is 2. In the other it is 3. Deep binding makes sure that each function reference uses the proper instance.

(c) (4 points) Here is another function that does the same thing:

\[
\begin{align*}
\text{define P2} & \quad \text{(lambda (n)} \nonumber \\
\text{(lambda (x)} & \nonumber \\
\text{(letrec ((F (lambda (s k) (if (= k 0) s (F (* s x) (- k 1))))) (F 1 n)))}) & \nonumber 
\end{align*}
\]

Explain why we might prefer to use this version.

Answer: It’s tail recursive. In any well-designed language implementation, it will be faster and more space efficient.

14. (Extra Credit; 8 points max) Some languages (e.g., Python) allow the programmer to omit variable declarations entirely. Other languages (e.g., Scheme) require the programmer to declare all variables, but not to declare their types. Still other languages (e.g., Java) require the programmer to declare both variables and their types.

Give a short (1-2 sentence) argument in favor of each approach. Which argument do you find most convincing? Why?

Answer: The Python approach leads to shorter programs that are arguably easier to write and read. It provides polymorphism “for free,” and relieves the programmer of the need to prove to the compiler that the program is type safe.

The Scheme approach is naturally polymorphic, but avoids errors due to misspellings and to accidental use of the same variable name for multiple purposes.

The Java approach is self-documenting, and arguably forces the programmer to think through things more carefully, reducing the incidence of bugs. Bugs that do occur are likely to be found at compile time. Given detailed knowledge of types, the compiler can produce more efficient code than it can for Scheme or Python.

Which approach is better is largely a matter of personal preference. Experience suggests, however, that very large systems are easier to maintain (and run much faster) when written in statically typed languages.