CS 401/501 EXAM #1 SAMPLE QUESTIONS (SOLUTION)

1. The string representation of a token is called a(n) `lexeme`.

2. True or False. Context-sensitive syntax refers to that part of a context-free grammar that requires nondeterminism to be parsed.

   ```
   False. Context-sensitive syntax is the same as static semantics,
   which means the correctness of the syntax with respect to context,
   such as the symbol table, e.g. type compatibility of left and right
   sides of an assignment statement.
   ```

3. List the phases of a *compiler*. Discuss in detail what each phase does and how the phases interact with each other. (You may draw a diagram to assist in this discussion if you wish.) Explain how an *interpreter* is different from a compiler.

   Described in section 1.71-1.7.3 of the textbook.

4. A C++ class declaration is described in Extended Backus-Naur form as:

   class-specifier ::=
       class identifier [**:** identifier] { member-declaration {member-declaration} }
   member-declaration ::= identifier identifier function-definition
       | **int** identifier function-definition

   Note that this syntax contains terminal symbols { and } which are different from the extended BNF meta-symbols { and }, and distinguished by underlining. Write an equivalent syntax graph and a recursive descent parser for this construction. Assume the existence of production rules for function-definition and corresponding parsing procedure `FUNCTION_DEFINITION`, and a scanning procedure `GET_TOKEN` to set a global string variable `TOKEN` to the token type of the next token. Furthermore, assume that `TOKEN` is already set to the first token when parsing begins.

   ```
   The syntax graph is shown in
   http://www.cis.uab.edu/cs401/spring2009/exam1topicssolfig.pdf.
   ```

The recursive descent parser is:

```
procedure CLASS_SPECIFIER;
  begin
    if TOKEN != CLASS then ERROR;
    GET_TOKEN;
    if TOKEN != IDENTIFIER then ERROR;
    GET_TOKEN;
    if TOKEN = COLON then begin
      GET_TOKEN;
      if TOKEN != IDENTIFIER then ERROR;
      GET_TOKEN;
    end;
    if TOKEN != LEFT_BRACE then ERROR;
    GET_TOKEN;
    repeat
      MEMBER_DECLARATION;
    until TOKEN = RIGHT_BRACE;
    GET_TOKEN;
  end;

procedure MEMBER_DECLARATION;
  begin
    if TOKEN = IDENTIFIER then
      GET_TOKEN;
    else if TOKEN = INT then
      GET_TOKEN;
    else
      ERROR;
    if TOKEN != IDENTIFIER then ERROR;
    GET_TOKEN;
    FUNCTION_DEFINITION;
  end;
```

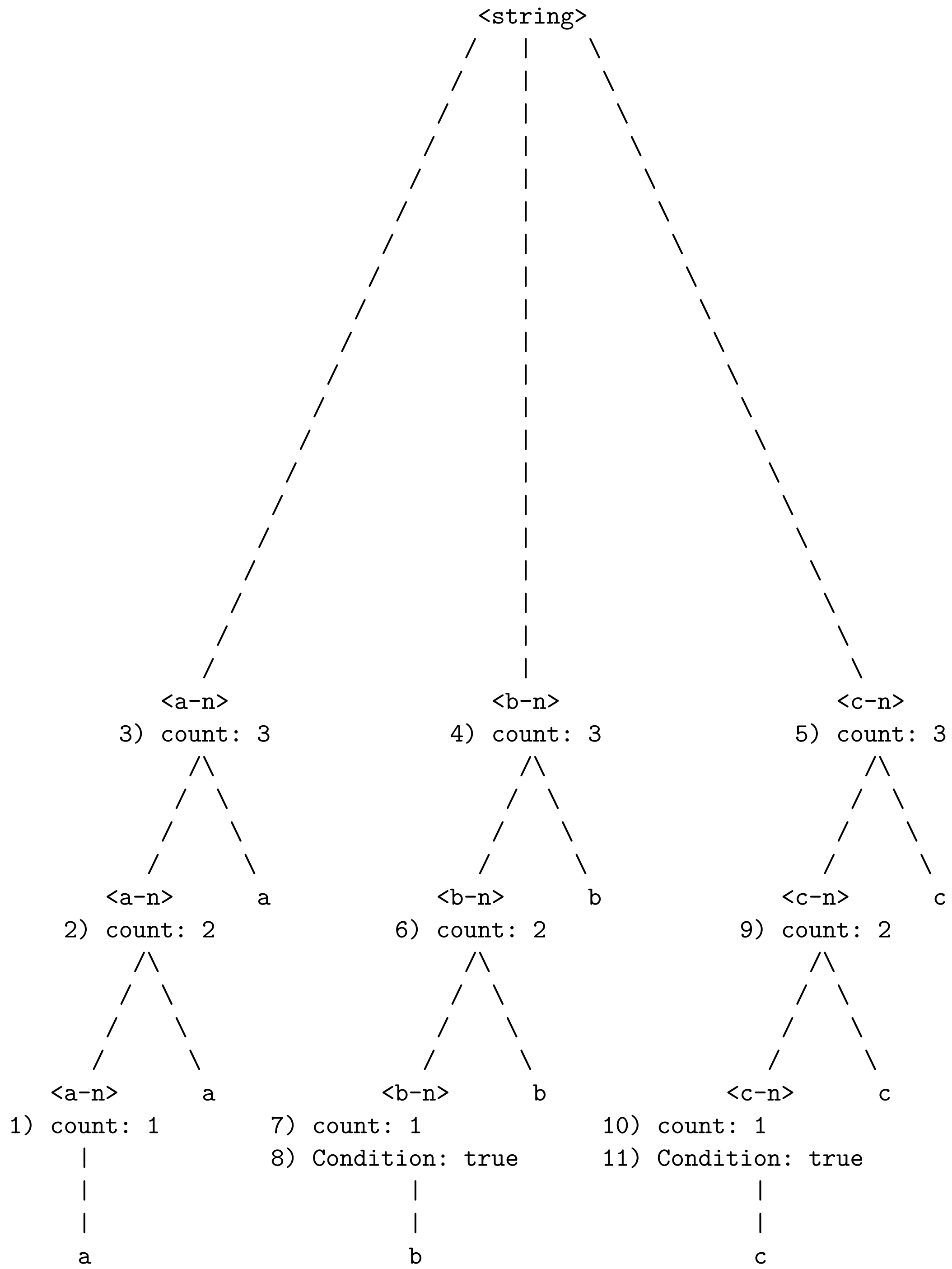
5. Consider the attribute grammar below:

<string> ::= <a-n> <b-n> <c-n>
        <b-n> . count ← <a-n> . count
        <c-n> . count ← <a-n> . count
<a-n> ::= a
        <a-n> . count ← 1
    | <a-n>$_1$ a
        <a-n> . count ← <a-n>$_1$. count + 1
<b-n> ::= b
        Condition: <b-n> . count = 1
    | <b-n>$_1$ b
        <b-n>$_1$. count ← <b-n> . count − 1
<c-n> ::= c
        Condition: <c-n> . count = 1
    | <c-n>$_1$ c
        <c-n>$_1$. count ← <c-n> . count − 1

(a) Indicate which attributes are inherited, which are synthesized, and which are intrinsic.

```
inherited - count of <b-n> and <c-n>
synthesized - count of <a-n>
intrinsic - none
```

(b) Draw the attributed parse tree for the string “aaabbbccc”, showing clearly the order of evaluation of all attributes.



<string>
<a-n>
3) count: 3
<b-n>
4) count: 3
<c-n>
5) count: 3
<a-n>
a
2) count: 2
<b-n>
b
6) count: 2
<c-n>
c
9) count: 2
<a-n>
a
1) count: 1
<b-n>
b
7) count: 1
8) Condition: true
<c-n>
c
10) count: 1
11) Condition: true
a
b
c

6. In pure λ-calculus, integers, Boolean values, lists, etc., must all be represented by λ-expressions. For example, $0 = \lambda x.\lambda y.y$, $1 = \lambda x.\lambda y.x\ y$, $2 = \lambda x.\lambda y.x\ (x\ y)$, etc., and $succ = \lambda z.\lambda x.\lambda y.x\ ((z\ x)\ y)$. *succ* is a function which adds 1 to its argument. Evaluate *succ* 1. Did you get 2? Why or why not?

   succ 1 $= (\lambda \text{z}.\lambda \text{x}.\lambda \text{y}.\text{x}\ ((\text{z}\ \text{x})\ \text{y}))(\lambda \text{x}.\lambda \text{y}.\text{x}\ \text{y}) \Rightarrow \lambda \text{x}.\lambda \text{y}.\text{x}\ (((\lambda \text{x}.\lambda \text{y}.\text{x}\ \text{y})\ \text{x})\ \text{y})$

   Technically, the reduction stops here as there are no actual parameters corresponding to x. However, by inspecting the inside expression, there is an internal redex which may be reduced further as follows (including further reductions of internal redexes).

   $\lambda \text{x}.\lambda \text{y}.\text{x}\ (((\lambda \text{x}.\lambda \text{y}.\text{x}\ \text{y})\ \text{x})\ \text{y}) \Rightarrow \lambda \text{x}.\lambda \text{y}.\text{x}\ ((\lambda \text{y}.\text{x}\ \text{y})\ \text{y}) \Rightarrow \lambda \text{x}.\lambda \text{y}.\text{x}\ ((\lambda \text{y}.\text{x}\ \text{y})\ \text{y}) \Rightarrow \lambda \text{x}.\lambda \text{y}.\text{x}\ (\text{x}\ \text{y}) =$ 2