CMSC 330 Using grammars for programming languages–solutions Fall 2012

1. There are different ways to write the grammars correctly.

   (Long nonterminal names, like those used in some of the solutions here, are fine in general and can be helpful since they are more descriptive, but in an exam situation you might prefer to use short single–letter nonterminal names in order to be able to write answers more quickly.)

   a.
   ```
   <binary−formula> ::= <unary−formula> → <binary−formula> | <unary−formula>
   <unary−formula>  ::= ∼<unary−formula> | ( <binary−formula> ) | <operand>
   <operand>        ::= p | q
   ```

   b.
   ```
   <binary−expr>  ::= <binary−expr> -> <unary−expr> | <unary−expr>
   <unary−expr>   ::= * <unary−expr> | <postfix−expr>
   <postfix−expr> ::= <postfix−expr> ++ | <id>
   <id>           ::= a | b
   ```

   c.
   ```
   <apl−expr> ::= + <apl−expr> | − <apl−expr> | ∗ <apl−expr> | / <apl−expr> |
                  <operand> + <apl−expr> | <operand> − <apl−expr> |
                  <operand> ∗ <apl−expr> | <operand> / <apl−expr> | <operand>
   <operand>  ::= a | b | c | ( <apl−expr> )
   ```

   d. Note that the first vertical bar in the productions for S is the | operator, not the vertical bar separating the right sides of the different productions for S.

   ```
   S → S | T | T
   T → T ^ U | U
   U → U & V | V
   V → V « W | V » W | W
   W → ~W | X
   X → n | (S)
   ```

   e.
   ```
   S → T | S | T
   T → U ^ T | U
   U → V & U | V
   V → W « V | W » V | W
   W → ~W | X
   X → n | (S)
   ```

f.
```
S → S | T | T
T → T ^ U | U
U → U & V | V
V → V << Y | V >> Y | W
W → ~W | X
X → n | (S)
Y → 0xF | 0x1
```

g.
```
S → aT
T → T[n] | T[S] | [n] | [S]
```

h.
```
S → S < V | S > V | S == V | T
T → x = T | y = T | z = T | U
U → V ^ U | V
V → (S) | x | y | z | x++ | y++ | z++
```

2. Both these grammars cure the ambiguity. The first uses the `end` ending an `if` statement to allow an `else` to be correctly associated with the proper `if` statement.

   The second grammar also cures the problem, by requiring the `then` part of the `if` statement to be surrounded by a `begin`/`end` pair.

   The best way to see this is to try derivations with grammar of some string that has an `if` statement containing a nested `if` statement with an `else` part, and convince yourself there is only one derivation possible for it in each of these grammars.

   Note that these grammars solve the dangling `else` problem by requiring that extra keywords (`end` or `begin` and `end`) be added to some or all `if` statements.

3. a.
```
<start>  ::= ( define f ( <mid> )
<mid>    ::= <formal> <mid> <actual> | ) <body> ) ( f
<formal> ::= <id>
<actual> ::= <expr>
<body>   ::= ...
<id>     ::= ...
<expr>   ::= ...
```

   The idea behind the grammar is that it matches the first formal parameter in the function's definition with the last actual parameter in the function's call, the second formal parameter in the function's definition with the second–to–last actual parameter in the call, etc. You can see that it works by deriving a program in which the function `f` has no parameters and the call has no arguments, a program in which the function `f` has one parameter and the call has one argument, and a program in which the function `f` has several parameters and the call has several arguments.

   b. This approach wouldn't work correctly for more than one function call. As a matter of fact, it is not possible to do this sort of match with a context free grammar (CFG), since a CFG cannot match arbitrarily many symbols. For example, although we will not prove this, the language $\mathrm{L} = \{\, a^{\mathrm{n}} b^{\mathrm{n}} c^{\mathrm{n}} \mid \mathrm{n} \geq 0 \,\}$ is not a context–free language (if you tried, you could never come up with a CFG generating it). And this is the formal language that essentially describes matching exactly two function calls with one function definition.

So although grammars are used to describe programming–language syntax when writing a compiler, some syntactic requirements must be enforced through some means other than a grammar. In fact, a later phase of a compiler checks properties like this that can't be captured in a CFG.