Module - 2

We'll expand on the grammer defined in Module - 1. The grammer for this module is exactly the same as the one before, except the addition of the ability to declare multiple variables on a single line. In effect, the grammer subset you have to implement is:

⟨program⟩ → class Program '{' ⟨field_decl⟩∗ ⟨statement_decl⟩∗ '}'

⟨field_decl⟩ → ⟨type⟩ {⟨id⟩ | ⟨id⟩ '[' ⟨int_literal⟩ ']' }+, ;

⟨statement_decl⟩ → ⟨location⟩ ⟨assign_op⟩ ⟨expr⟩ ;
| callout (⟨string_literal⟩ [, ⟨callout_arg⟩+,]) ;

The primary task in this module is to build the abstract syntax tree (AST) for the parsed program. Once we have the AST, we can do a variety of operations on it, including traversals(DFS/BFS), evaluations and conversions.
You have complete freedom on how you build the AST, but **your output must match the given output** (which can easily be ensured using a proper traversal of your AST).

We'll use two traversals to check the construction of your AST. The first traversal should print your AST, while the second one will evaluate the value at each of the interesting nodes.

Output to stdout, roll number of first team-mate on a single line.
Then output the role number of second team-mate (or 0 if not applicable) on second line.
Then output "Success" on a successful parse, and "Syntax Error" in case of an error, on the third line.

On a successful parse, make the following output files:

XML_visitor.txt : First Traversal

The first traversal will in-effect convert your AST to XML, and output it. The table below will explain the mappings between the two.

Output file: XML_visitor.txt

program	<program> <field_declarations count="n"> ... </field_declarations> <statement_declarations count="m"> … </statement_declarations> </program> n : number of named field declarations

<table>
<tr><td></td><td>m : number of statements</td></tr>
<tr><td>field_declaration
(will go inside <field_declarations>)</td><td>Normal
<declaration name=”x” type=”t” />

Array
<declaration name=”x” count=”n” type=”t” />

x : Name of the variable
n : Number of elements
t : type (“integer” / “boolean”)</td></tr>
<tr><td>callout
(will go inside <statement_declarations>)</td><td><callout function=”x”>
…
… Arguments to Callout …
…
</callout></td></tr>
<tr><td>assignment
(will go inside <statement_declarations>)</td><td><assignment>
<location … />
… expr ...
</assignment></td></tr>
<tr><td>location</td><td>Normal
<location id=”x” />

Array
<s><location id=”x” position=”n“ /></s>

<location id=”x”>
<position>
… expr …
</position>
</location></td></tr>
<tr><td>int literal</td><td><integer value=”n” /></td></tr>
<tr><td>char literal</td><td><character value=”x” /></td></tr>
<tr><td>bool literal</td><td><boolean value=”true/false” /></td></tr>
<tr><td>string</td><td><string value=”x” /></td></tr>
<tr><td>l.h.expr bin_op r.h.expr</td><td><binary_expression type=”x”>
… left hand expr …
… right hand expr …
</binary_expression>

<table><tr><td>bin_op</td><td>x</td></tr></table></td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>‘+’</td><td>addition</td></tr>
<tr><td>‘-’</td><td>subtraction</td></tr>
<tr><td>‘*’</td><td>multiplication</td></tr>
<tr><td>‘/’</td><td>division</td></tr>
<tr><td>‘%’</td><td>remainder</td></tr>
<tr><td>‘<’</td><td>less_than</td></tr>
<tr><td>‘>’</td><td>greater_than</td></tr>
<tr><td>‘<=’</td><td>less_equal</td></tr>
<tr><td>‘>=’</td><td>greater_equal</td></tr>
<tr><td>‘==’</td><td>is_equal</td></tr>
<tr><td>‘!=’</td><td>is_not_equal</td></tr>
<tr><td>‘&&’</td><td>and</td></tr>
<tr><td>‘||’</td><td>or</td></tr>
</table>
</td></tr>
<tr><td>un_op expr</td><td>
<unary_expression type=”x”>
… expr …
</unary_expression>

<table>
<tr><td>un_op</td><td>x</td></tr>
<tr><td>-</td><td>minus</td></tr>
<tr><td>!</td><td>not</td></tr>
</table>
</td></tr>
</table>