Decaf Project Module #3 – AST Deadline: Oct 12, 2016

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).

The table below contains a mapping between rules of the grammar and the output to be printed by their corresponding AST nodes.

Output to stdout:

- "Success" on a successful parse
- "Syntax Error" in case of an error

On a successful parse, make an output file "XML_visitor.txt".

The table below will explain the mappings for the following rules. However, you have to implement it for the entire Decaf grammar, in a similar fashion.

⟨ 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⟩ +,]) ;

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 m : number of statements
field_declaration (will go inside <field_declarations>)	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”)
callout (will go inside <statement_declarations>)	<callout function=”x”> … … Arguments to Callout … … </callout>
assignment (will go inside <statement_declarations>)	<assignment> <location … /> … expr ... </assignment>
location	Normal <location id=”x” /> Array ~~<location id=”x” position=”n“ />~~ <location id=”x”> <position> … expr … </position> </location>
int literal	<integer value=”n” />
char literal	<character value=”x” />
bool literal	<boolean value=”true/false” />
string	<string value=”x” />
l.h.expr bin_op r.h.expr	<binary_expression type=”x”> … left hand expr … … right hand expr … </binary_expression>

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

Submission format:

Compress a) the flex code (named Module3.l), b) the bison code (named Module3.y), c) other files, d) a readme file, e) a executable (named Module3), and f) a makefile and upload the zip file. The output file generated must be as specified above.

The zip file should be named rollno1_rollno2_Module3.zip.