```
 1 /*
 2  *  Markup.h
 3  *  Defines the markup class, which represents a parse tree of the code
 4  *
 5  *
 6  *  Created: 1/10/2017 by Ryan Tedeschi
 7  */
 8
 9 #ifndef MARKUP_H
10 #define MARKUP_H
11
12 #include <vector>
13 #include <string>
14 #include <regex>
15 #include <iostream>
16 #include "../Helpers/Helpers.h"
17 #include "../Printable/Printable.h"
18
19 /*
20     The Markup class is used to represent the parse tree of a particular snippet of code.
21 */
22 class Markup : public Printable
23 {
24     public:
25         /*
26             Creates a Markup object with no ID or Data
27         */
28         Markup();
29         /*
30             Creates a Markup object with only an ID. This is meant for production nodes
31             id - accociated production/terminal title
32         */
33         Markup(string id);
34         /*
35             Creates a Markup object with both an ID and Data. This is meant for terminal nodes (leaves)
36             id - associated production/terminal title
37             data - code associated with the node
38         */
39         Markup(string id, string data);
40         /*
41             Destructor (UNIMPLEMENTED)
42         */
43         ~Markup();
44
45         /*
46             Adds a child to the end of the markup list
47             c - child to add
48         */
49         void AddChild(Markup* c);
50         /*
51             Concatenates a vector of children to the end of thet markup list
52             list - vector of children to add
53         */
54         void AddChildren(vector<Markup*> list);
55         /*
56             Retrieves the child at the specified index.
57             i - if non-negative, indexes from the front of the child array. If negative, indexes from the back of the child array
58         */
59         Markup* ChildAt(int i);
60         /*
61             Retrieves a vector containing recursive productions matching the current ID.
62         */
63         vector<Markup*> RecursiveElements();
64
65         /*
66             Finds the first matching child by ID, null if no match
```

```
67              id - ID to match
68          */
69          Markup* FindFirstChildById(string id);
70          /*
71              Finds the first matching node in self or any descendants by ID, null if no match
72              id - ID to match
73          */
74          Markup* FindFirstById(string id);
75          /*
76              Finds all matching children by ID
77              id - ID to match
78          */
79          vector<Markup*> FindAllChildrenById(string id);
80          /*
81              Finds all matching self or descendants by ID
82              id - ID to match
83              findChildrenOfMatches - if true, continues searching inside matching nodes
84          */
85          vector<Markup*> FindAllById(string id, bool findChildrenOfMatches);
86
87          /*
88              Finds the first matching terminal by value, null if no match
89              id - optional terminal ID
90              val - value to match
91          */
92          Markup* FindFirstTerminalByVal(string id, string val);
93          Markup* FindFirstTerminalByVal(string val);
94          /*
95              Finds all matching terminals by value
96              id - optional terminal ID
97              val - value to match
98          */
99          vector<Markup*> FindAllTerminalsByVal(string id, string val);
100         vector<Markup*> FindAllTerminalsByVal(string val);
101         /*
102             Finds the first matching ancestor by ID, NULL if no match
103             id - ID to match
104         */
105         Markup* FindAncestorById(string id);
106
107         /*
108             Retrieves the parent of the node (NULL if none)
109         */
110         Markup* Parent();
111         /*
112             Retrieves the number of children of the node
113         */
114         int NumChildren();
115         /*
116             Retrieves the associated code of the node. If the node is a leaf,
117             this returns the string data. Otherwise, this collects all leaf data and returns it
118         */
119         string GetData();
120         /*
121             Retrieves the associated production/terminal ID
122         */
123         string GetID();
124         /*
125             Retrieves the vector of children
126         */
127         vector<Markup*> Children();
128         /*
129             Returns if this node is a root (no parent)
130         */
131         bool IsRoot();
132         /*
133             Returns if this node is a leaf (no children)
```

```
134         */
135         bool IsLeaf();
136 
137         /*
138             Prints the node out
139         */
140         void Print();
141         /*
142             Prints the node out at a specific tab indent
143         */
144         void Print(int tabIndex);
145         /*
146             Gets the index of the node in its parent
147         */
148         int IndexInParent();
149         /*
150             Gets all accessible variable declarations to the node
151         */
152         unordered_map<string, string> AccessibleDeclarations();
153         /*
154             Gets all accessible variable declarations to the node
155         */
156         unordered_map<string, Markup*> AccessibleValues();
157 
158         unordered_map<string, string> localDeclarations;
159         unordered_map<string, Markup*> localValues;
160 
161         /*
162             Deep copies the object
163         */
164         Markup* Clone();
165 
166     private:
167         // Parent of the node
168         Markup* parent;
169         // list of children of the node
170         vector<Markup*> children;
171         // code data - only used in leaf nodes
172         string data;
173         // production/terminal ID
174         string id;
175         // index in parent
176         int index = 0;
177 };
178 
179 #endif
```