```cpp
/*
 *  Markup.cpp
 *  Defines the markup class, which is the hierarchical representation of code
 *
 *
 *  Created: 1/10/2017 by Ryan Tedeschi
 */

#include "Markup.h"

Markup::Markup() {
    parent = NULL;
}
Markup::Markup(string id, string data) {
    parent = NULL;
    this->data = data;
    this->id = id;
}
Markup::Markup(string id) {
    parent = NULL;
    this->id = id;
}
Markup::~Markup() {

}

void Markup::AddChild(Markup* c) {
    c->parent = this;
    c->index = children.size();
    children.push_back(c);
}

void Markup::AddChildren(vector<Markup*> list) {
    for (int i = 0; i < list.size(); i++) {
        Markup* c = list[i];
        c->parent = this;
        c->index = children.size();
        children.push_back(c);
    }
}

int Markup::NumChildren() {
    return children.size();
}
Markup* Markup::ChildAt(int i) {
    if (i >= 0) {
        if (i < children.size())
            return children[i];
    } else {
        if (children.size() + i >= 0);
            return children[children.size() + i];
    }
    return NULL;
}
Markup* Markup::Parent() {
    return parent;
}
vector<Markup*> Markup::Children() {
    return children;
}
string Markup::GetData() {
    if (!IsLeaf()) {
        string d = children[0]->GetData();
        for (int i = 1; i < children.size(); i++) {
            d += " " + children[i]->GetData();
        }
```

```
67          return d;
68      } else {
69          return data;
70      }
71  }
72
73  vector<Markup*> Markup::RecursiveElements() {
74
75      Markup* rm = this;
76      vector<Markup*> recursives;
77
78      while (rm != NULL) {
79          recursives.push_back(rm);
80          rm = rm->FindFirstChildById(id);
81      }
82
83      return recursives;
84  }
85
86  string Markup::GetID() {
87      return id;
88  }
89
90  bool Markup::IsRoot() {
91      return parent == NULL;
92  }
93  bool Markup::IsLeaf() {
94      return children.size() == 0;
95  }
96
97  void Markup::Print() {
98      Print(0);
99  }
100 void Markup::Print(int tabIndex) {
101     int i;
102     for (i = 0; i < tabIndex; i++)
103         cout << "\t";
104
105     cout << id << ": \"" << GetData() << "\"\n";
106
107     for (int i = 0; i < NumChildren(); i++) {
108         children[i]->Print(tabIndex + 1);
109     }
110 }
111
112 Markup* Markup::FindFirstById(string id) {
113     Markup* result = NULL;
114     if (this->id == id) {
115         result = this;
116     } else {
117         for (int i = 0; i < children.size(); i++) {
118             if ((result = children[i]->FindFirstById(id)) != NULL)
119                 break;
120         }
121     }
122     return result;
123 }
124 vector<Markup*> Markup::FindAllById(string id, bool findChildrenOfMatches) {
125     vector<Markup*> results;
126
127     if (this->id == id) {
128         results.push_back(this);
129     }
130
131     if (this->id != id || findChildrenOfMatches) {
132         for (int i = 0; i < children.size(); i++) {
133             vector<Markup*> v = children[i]->FindAllById(id, findChildrenOfMatches);
```

```
134             results = Helpers::concat(results, v);
135         }
136     }
137 
138     return results;
139 }
140 
141 Markup* Markup::FindFirstChildById(string id) {
142     Markup* result = NULL;
143 
144     for (int i = 0; i < children.size(); i++) {
145         if (children[i]->id == id) {
146             result = children[i];
147             break;
148         }
149     }
150 
151     return result;
152 }
153 vector<Markup*> Markup::FindAllChildrenById(string id) {
154     vector<Markup*> results;
155 
156     for (int i = 0; i < children.size(); i++) {
157         if (children[i]->id == id)
158             results.push_back(children[i]);
159     }
160 
161     return results;
162 }
163 
164 Markup* Markup::FindFirstTerminalByVal(string id, string val) {
165     Markup* result = NULL;
166     if (this->IsLeaf()) {
167         if (this->id == id && this->data == val)
168             result = this;
169     } else {
170         for (int i = 0; i < children.size() && result == NULL; i++) {
171             result = children[i]->FindFirstTerminalByVal(id, val);
172         }
173     }
174     return result;
175 }
176 Markup* Markup::FindFirstTerminalByVal(string val) {
177     Markup* result = NULL;
178     if (this->IsLeaf()) {
179         if (this->data == val)
180             result = this;
181     } else {
182         for (int i = 0; i < children.size() && result == NULL; i++) {
183             result = children[i]->FindFirstTerminalByVal(val);
184         }
185     }
186     return result;
187 }
188 vector<Markup*> Markup::FindAllTerminalsByVal(string id, string val) {
189     vector<Markup*> results;
190     if (this->IsLeaf()) {
191         if (this->id == id && this->data == val)
192             results.push_back(this);
193     } else {
194         for (int i = 0; i < children.size(); i++) {
195             vector<Markup*> v = children[i]->FindAllTerminalsByVal(val);
196             results = Helpers::concat(results, v);
197         }
198     }
199     return results;
200 }
```

```
201 vector<Markup*> Markup::FindAllTerminalsByVal(string val) {
202     vector<Markup*> results;
203     if (this->IsLeaf()) {
204         if (this->data == val)
205             results.push_back(this);
206     } else {
207         for (int i = 0; i < children.size(); i++) {
208             vector<Markup*> v = children[i]->FindAllTerminalsByVal(val);
209             results = Helpers::concat(results, v);
210         }
211     }
212     return results;
213 }
214
215 Markup* Markup::FindAncestorById(string id) {
216     Markup* result = NULL;
217     if (parent != NULL) {
218         if (parent->id == id)
219             result = parent;
220         else
221             result = parent->FindAncestorById(id);
222     }
223     return result;
224 }
225
226 unordered_map<string, string> Markup::AccessibleDeclarations() {
227     unordered_map<string, string> declarations;
228
229     // try to get any global declarations
230     Markup* statementAncestor = FindAncestorById("statement");
231     if (statementAncestor != NULL) {
232         unordered_map<string, string> parentDecl = statementAncestor->AccessibleDeclarations();
233         for ( auto it = parentDecl.begin(); it != parentDecl.end(); ++it )
234             declarations[it->first] = it->second;
235     }
236
237     Markup* fnAncestor = FindAncestorById("function-definition");
238     if (fnAncestor != NULL) {
239         unordered_map<string, string> parentDecl = fnAncestor->AccessibleDeclarations();
240         for ( auto it = parentDecl.begin(); it != parentDecl.end(); ++it )
241             declarations[it->first] = it->second;
242     }
243
244     // try to get previous sibling declarations
245     if (parent != NULL) {
246         for (int i = 0; i < index; i++) {
247             unordered_map<string, string> sibDecl = parent->ChildAt(i)->AccessibleDeclarations();
248             for ( auto it = sibDecl.begin(); it != sibDecl.end(); ++it )
249                 declarations[it->first] = it->second;
250         }
251     }
252
253     // try to get previous statement declarations
254     if (parent != NULL && id == "statement") {
255         vector<Markup*> s;
256         Markup* sl = parent;
257         while ((sl = sl->parent) != NULL && sl->GetID() == "statement-list") {
258             s.insert(s.begin(), sl->FindFirstChildById("statement"));
259         }
260         for (int i = 0; i < s.size(); i++) {
261             unordered_map<string, string> sibDecl = s[i]->AccessibleDeclarations();
262             for ( auto it = sibDecl.begin(); it != sibDecl.end(); ++it )
263                 declarations[it->first] = it->second;
264         }
265     }
266
267     // add any local declarations
```

```
268     for ( auto it = localDeclarations.begin(); it != localDeclarations.end(); ++it )
269         declarations[it->first] = it->second;
270 
271     return declarations;
272 }
273 
274 unordered_map<string, Markup*> Markup::AccessibleValues() {
275     unordered_map<string, Markup*> values;
276 
277     // cout << "Getting accessible values on " << GetData() << " (" << GetID() << ")" << endl;
278 
279     // try to get any global values
280     Markup* statementAncestor = FindAncestorById("statement");
281     if (statementAncestor != NULL) {
282         unordered_map<string, Markup*> parentDecl = statementAncestor->AccessibleValues();
283         for ( auto it = parentDecl.begin(); it != parentDecl.end(); ++it )
284             values[it->first] = it->second;
285     }
286 
287     Markup* fnAncestor = FindAncestorById("function-definition");
288     if (fnAncestor != NULL) {
289         unordered_map<string, Markup*> parentDecl = fnAncestor->AccessibleValues();
290         for ( auto it = parentDecl.begin(); it != parentDecl.end(); ++it )
291             values[it->first] = it->second;
292     }
293 
294     // try to get previous sibling values
295     if (parent != NULL) {
296         for (int i = 0; i < index; i++) {
297             unordered_map<string, Markup*> sibDecl = parent->ChildAt(i)->AccessibleValues();
298             for ( auto it = sibDecl.begin(); it != sibDecl.end(); ++it )
299                 values[it->first] = it->second;
300         }
301     }
302 
303     // try to get previous statement values
304     if (parent != NULL && id == "statement") {
305         vector<Markup*> s;
306         Markup* sl = parent;
307         while ((sl = sl->parent) != NULL && sl->GetID() == "statement-list") {
308             s.insert(s.begin(), sl->FindFirstChildById("statement"));
309         }
310         for (int i = 0; i < s.size(); i++) {
311             unordered_map<string, Markup*> sibDecl = s[i]->AccessibleValues();
312             for ( auto it = sibDecl.begin(); it != sibDecl.end(); ++it )
313                 values[it->first] = it->second;
314         }
315     }
316 
317     // add any local values
318     for ( auto it = localValues.begin(); it != localValues.end(); ++it )
319         values[it->first] = it->second;
320 
321     return values;
322 }
323 
324 int Markup::IndexInParent() {
325     return index;
326 }
327 
328 Markup* Markup::Clone() {
329     Markup* m = new Markup(id);
330     m->localDeclarations = localDeclarations;
331     m->localValues = localValues;
332 
333     if (IsLeaf()) {
334         m->data = data;
```

```
335     } else {
336         for (int i = 0; i < children.size(); i++) {
337             m->AddChild(children[i]->Clone());
338         }
339     }
340     return m;
341 }
```