```cpp
#include "OutlineModule.h"

static string _OutlineModule = RegisterPlugin("Outline", new OutlineModule());

string EntryTypes[] = { "Start", "MethodCall", "Process", "Loop", "Decision", "EndDecision", "IO", "End" };

OutlineModule::OutlineModule() {}

CASP_Return* OutlineModule::Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn) {
    returnData = (inputReturn != NULL ? inputReturn : new CASP_Return());

    /*
        This module hasn't implemented any Function Args yet!
        Use Helpers::ParseArrayArgument() and Helpers::ParseArgument() to scrape out arguments
    */

    cout << "This is the entry point for the " << _OutlineModule << " Module!\n";

    // markup->Print();
    // vector<Markup*> m = markup->FindAllById("statement", false);
    // for (int i = 0; i < m.size(); i++) {
    //     m[i]->Print();
    // }

    vector<Outline*> outlines = GetAllOutlines(markup);
    return FormatData(outlines);
}

vector<Outline*> OutlineModule::GetAllOutlines(Markup* masterTree) {
    vector<Outline*> outlines;
    vector<Markup*> functions = masterTree->FindAllById("function-definition", true);
    vector<Markup*> sls = masterTree->FindAllChildrenById("statement-list");

    if (sls.size() > 0) {
        outlines.push_back(GetRootOutline(sls));
    }
    if (functions.size() > 0) {
        for (int i = 0; i < functions.size(); i++) {
            outlines.push_back(GetFunctionOutline(functions[i]));
        }
    }

    return outlines;
}

Outline* OutlineModule::GetRootOutline(vector<Markup*> sls) {
    string functionTitle = "ROOT";

    Outline* outline = new Outline();
    Node* currentNode = outline->AppendBlock(Start, functionTitle, NULL);

    for (int i = 0; i < sls.size(); i++) {
        Markup* block = new Markup();
        block->AddChild(sls[i]);
        currentNode = processBlock(block, outline, currentNode);
    }

    outline->AppendBlock(End, "End " + functionTitle, currentNode);
    return outline;
}

Outline* OutlineModule::GetFunctionOutline(Markup* functionTree) {
    string functionTitle = functionTree->FindFirstChildById("function-identifier")->GetData();
    Markup* declarationList = functionTree->FindFirstChildById("function-parameters")->FindFirstChildById("function-parameter-list");
    string startText = functionTitle;
    if (declarationList != NULL) {
```

```
67          startText += ": ";// + declarationList->GetData();
68          vector<Markup*> dls = declarationList->FindAllById("function-parameter-declaration", false);
69          for (int i = 0; i < dls.size(); i++) {
70              startText += "\n" + dls[i]->GetData();
71          }
72
73      }
74
75      Outline* outline = new Outline();
76      Node* currentNode = outline->AppendBlock(Start, startText, NULL);
77
78      Markup* block = functionTree->FindFirstById("block");
79      currentNode = processBlock(block, outline, currentNode);
80
81      outline->AppendBlock(End, "End " + functionTitle, currentNode);
82
83      return outline;
84  }
85
86  CASP_Return* OutlineModule::FormatData(vector<Outline*> outlines) {
87      GenericObject* data = returnData->Data();
88      GenericArray* o = new GenericArray();
89
90      for (int i = 0; i < outlines.size(); i++) {
91          o->Add(outlines[i]->Output());
92          outlines[i]->Print();
93          cout << endl;
94      }
95
96      data->Add("Outlines", o);
97
98      // ret->Print();
99      // cout << endl;
100
101     return returnData;
102 }
103
104 Node* OutlineModule::stripProcess(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
105
106     Node* currentNode = startNode;
107
108     string type = parseTree->GetID();
109     bool sameType = currentNode->data.find(type + ":") == 0;
110
111     if (!sameType) {
112         currentNode = outline->AppendBlock(Process, type + ":\n\t" + parseTree->GetData(), currentNode, firstEdgeData);
113     } else {
114         currentNode->data += "\n\t" + parseTree->GetData();
115     }
116
117     // cout << parseTree->GetID() << endl;
118     // parseTree->Print();
119
120     return currentNode;
121 }
122 Node* OutlineModule::stripMethodCall(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
123     string blockData = parseTree->FindFirstChildById("function-identifier")->GetData();
124     Markup* methodArgsTree = parseTree->FindFirstChildById("arg-list");
125
126     if (methodArgsTree != NULL) {
127         blockData = blockData + ": " + methodArgsTree->GetData();
128     }
129
130     return outline->AppendBlock(MethodCall, blockData, startNode, firstEdgeData);
131 }
132 Node* OutlineModule::stripDecision(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
133
```

```
134     Node* currentDecisionHead;
135     Node* currentNode = startNode;
136     Node* endDecision = new Node("End Decision", EndDecision, 0);
137 
138     Markup* condition = parseTree->FindFirstChildById("expression");
139     Markup* body = parseTree->FindFirstChildById("decision-body");
140     Markup* proc;
141     Markup* dc = parseTree->FindFirstChildById("decision-cases");
142     vector<Markup*> decisionCases;
143     while (dc != NULL) {
144         decisionCases.push_back(dc->FindFirstChildById("decision-case"));
145         dc = dc->FindFirstChildById("decision-cases");
146     }
147     Markup* fallback = parseTree->FindFirstChildById("decision-fallback");
148     string blockData;
149 
150     blockData = condition->GetData() + "?";
151     currentDecisionHead = outline->AppendBlock(Decision, blockData, currentNode, firstEdgeData);
152 
153     if ((proc = body->FindFirstChildById("block")) != NULL) {
154         currentNode = processBlock(proc, outline, currentDecisionHead, "True");
155         currentNode->AddEdgeTo(endDecision);
156     }
157     else if ((proc = body->FindFirstChildById("statement")) != NULL) {
158         currentNode = processStatement(proc, outline, currentDecisionHead, "True");
159         currentNode->AddEdgeTo(endDecision);
160     }
161     else {
162         currentNode->AddEdgeTo(endDecision, "True");
163     }
164 
165     for (int i = 0; i < decisionCases.size(); i++) {
166         condition = decisionCases[i]->FindFirstChildById("expression");
167         body = decisionCases[i]->FindFirstChildById("decision-body");
168         blockData = condition->GetData() + " ?";
169         currentDecisionHead = outline->AppendBlock(Decision, blockData, currentDecisionHead, "False");
170 
171         if ((proc = body->FindFirstChildById("block")) != NULL) {
172             currentNode = processBlock(proc, outline, currentDecisionHead, "True");
173             currentNode->AddEdgeTo(endDecision);
174         }
175         else if ((proc = body->FindFirstChildById("statement")) != NULL) {
176             currentNode = processStatement(proc, outline, currentDecisionHead, "True");
177             currentNode->AddEdgeTo(endDecision);
178         }
179         else {
180             currentNode->AddEdgeTo(endDecision, "True");
181         }
182     }
183 
184     if (fallback != NULL) {
185         body = fallback->FindFirstChildById("decision-body");
186         if ((proc = body->FindFirstChildById("block")) != NULL) {
187             currentNode = processBlock(proc, outline, currentDecisionHead, "False");
188             currentNode->AddEdgeTo(endDecision);
189         }
190         else if ((proc = body->FindFirstChildById("statement")) != NULL) {
191             currentNode = processStatement(proc, outline, currentDecisionHead, "False");
192             currentNode->AddEdgeTo(endDecision);
193         }
194         else {
195             currentNode->AddEdgeTo(endDecision, "False");
196         }
197     } else {
198         currentDecisionHead->AddEdgeTo(endDecision, "False");
199     }
200 
```

```
201     return outline->AppendBlock(endDecision);
202 }
203 Node* OutlineModule::stripFor(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
204 
205     Markup* init = parseTree->FindFirstChildById("for-init")->ChildAt(0);
206     Markup* condition = parseTree->FindFirstChildById("for-condition")->ChildAt(0);
207     Markup* increment = parseTree->FindFirstChildById("for-increment")->ChildAt(0);
208     Markup* body = parseTree->FindFirstChildById("for-body");
209     Markup* proc = NULL;
210     string blockData = "Loop";
211 
212     if (init != NULL || condition != NULL || increment != NULL) {
213         bool prev = false;
214         blockData += ": ";
215         if (init != NULL) {
216             blockData += init->GetData();
217             prev = true;
218         }
219         if (condition != NULL) {
220             if (prev)
221                 blockData += ", ";
222             blockData += condition->GetData();
223             prev = true;
224         }
225         if (increment != NULL) {
226             if (prev)
227                 blockData += ", ";
228             blockData += increment->GetData();
229         }
230     }
231 
232     Node* currentNode = startNode =
233         outline->AppendBlock(Loop, blockData, startNode, firstEdgeData);
234 
235     if ((proc = body->FindFirstChildById("block")) != NULL) {
236         currentNode = processBlock(proc, outline, startNode, "Loop Iteration");
237         currentNode->AddEdgeTo(startNode);
238     } else if ((proc = body->FindFirstChildById("statement")) != NULL) {
239         currentNode = processStatement(proc, outline, startNode, "Loop Iteration");
240         currentNode->AddEdgeTo(startNode);
241     } else {
242         currentNode->AddEdgeTo(currentNode, "Loop Iteration");
243     }
244 
245     return startNode;
246 }
247 Node* OutlineModule::stripWhile(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
248 
249     bool isDoWhile = parseTree->FindFirstChildById("DO") != NULL;
250     Markup* condition = parseTree->FindFirstChildById("while-condition")->ChildAt(0);
251     Markup* body = parseTree->FindFirstChildById("while-body");
252     Markup* proc = NULL;
253     string blockData = "Loop";
254 
255     if (condition != NULL) {
256         blockData += "\n" + condition->GetData() + "?";
257     } else {
258         blockData += "\n(no condition)";
259     }
260 
261     Node* currentNode = startNode =
262         outline->AppendBlock(Decision, blockData, startNode, firstEdgeData);
263 
264     if ((proc = body->FindFirstChildById("block")) != NULL) {
265         currentNode = processBlock(proc, outline, startNode, "Loop Iteration");
266         currentNode->AddEdgeTo(startNode);
267     } else if ((proc = body->FindFirstChildById("statement")) != NULL) {
```

```
268         currentNode = processStatement(proc, outline, startNode, "Loop Iteration");
269         currentNode->AddEdgeTo(startNode);
270     } else {
271         currentNode->AddEdgeTo(currentNode, "Loop Iteration");
272     }
273 
274     return startNode;
275 }
276 
277 Node* OutlineModule::processBlock(Markup* parseTree, Outline* outline, Node* startNode, string firstEdgeData) {
278     Node* currentNode = startNode;
279     Markup* csl = parseTree->FindFirstChildById("statement-list");
280     Markup* cs = NULL;
281     int ct = 0;
282 
283     while (csl != NULL) {
284         cs = csl->FindFirstChildById("statement");
285         currentNode = processStatement(cs, outline, currentNode, ct++ == 0 ? firstEdgeData : "");
286         csl = csl->FindFirstChildById("statement-list");
287     }
288     return currentNode;
289 }
290 Node* OutlineModule::processStatement(Markup* statement, Outline* outline, Node* startNode, string firstEdgeData) {
291     Node* currentNode = NULL;
292     Markup* s = statement->ChildAt(0);
293     string id = s->GetID();
294 
295     if (id == "for-loop") {
296         currentNode = stripFor(s, outline, startNode, firstEdgeData);
297     }
298     else if (id == "while-loop" || id == "do-while-loop") {
299         currentNode = stripWhile(s, outline, startNode, firstEdgeData);
300     } else if (id == "decision") {
301         currentNode = stripDecision(s, outline, startNode, firstEdgeData);
302     } else if (id == "block") {
303         currentNode = processBlock(s, outline, startNode, firstEdgeData);
304     } else if (id == "expression-statement") {
305         s = s->ChildAt(0)->ChildAt(0);
306         id = s->GetID();
307         while (id == "grouped-expression") {
308             s = s->ChildAt(1);
309             id = s->GetID();
310         }
311 
312         if (id == "method-invocation") {
313             currentNode = stripMethodCall(s, outline, startNode, firstEdgeData);
314         }
315         else {
316             currentNode = stripProcess(s, outline, startNode, firstEdgeData);
317         }
318     }
319 
320     return currentNode;
321 }
322 
323 Outline::Outline() {}
324 
325 GenericArray* Outline::Output() {
326     GenericArray* arr = new GenericArray();
327 
328     for (int i = 0; i < nodes.size(); i++) {
329         arr->Add(nodes[i]->Output());
330     }
331 
332     return arr;
333 }
334 
```

```
335 void Outline::Print() {
336     // if (head != NULL) {
337     //     head->Print();
338     // } else {
339     //     cout << "No data to print\n";
340     // }
341 
342     for (int i = 0; i < nodes.size(); i++) {
343         nodes[i]->Print();
344     }
345 }
346 
347 Node* Outline::AppendBlock(EntryType type, string nodeData, Node* sourceNode) {
348     return AppendBlock(type, nodeData, sourceNode, "");
349 }
350 
351 Node* Outline::AppendBlock(EntryType type, string nodeData, Node* sourceNode, string edgeData) {
352 
353     Node* node = new Node(nodeData, type, maxId++);
354     if (sourceNode != NULL) {
355         sourceNode->AddEdgeTo(node, edgeData);
356     }
357     if (head == NULL) {
358         head = node;
359     }
360     nodes.push_back(node);
361 
362     return node;
363 }
364 
365 Node* Outline::AppendBlock(Node* node) {
366 
367     node->id = maxId++;
368 
369     if (head == NULL) {
370         head = node;
371     }
372     nodes.push_back(node);
373 
374     return node;
375 }
376 
377 Node::Node(string data, EntryType type, int id) {
378 
379     this->id = id;
380     this->data = data;
381     this->type = type;
382 
383 }
384 
385 GenericObject* Node::Output() {
386     GenericObject* ob = new GenericObject();
387     GenericArray* arr = new GenericArray();
388 
389     ob->Add("id", CreateLeaf(id));
390     ob->Add("data", CreateLeaf(data));
391     ob->Add("type", CreateLeaf(EntryTypes[type]));
392 
393     for (int i = 0; i < edges.size(); i++) {
394         arr->Add(edges[i]->Output());
395     }
396 
397     ob->Add("edges", arr);
398 
399     return ob;
400 }
401 
```

```
402 void Node::Print() {
403     cout << id << "\t" << data << " (" << EntryTypes[type] << ")\n";
404     for (int i = 0; i < edges.size(); i++) {
405         cout << "\t" << (i + 1) << "\t";
406         edges[i]->Print();
407     }
408 
409     // for (int i = 0; i < edges.size(); i++) {
410     //     if (edges[i]->target->id > id)
411     //         edges[i]->target->Print();
412     // }
413 
414 }
415 
416 Edge* Node::AddEdgeTo(Node* toNode) {
417 
418     Edge* edge = new Edge(this, toNode);
419     edges.push_back(edge);
420 
421     return edge;
422 }
423 
424 Edge* Node::AddEdgeFrom(Node* fromNode) {
425 
426     Edge* edge = new Edge(fromNode, this);
427     fromNode->edges.push_back(edge);
428 
429     return edge;
430 }
431 
432 Edge* Node::AddEdgeTo(Node* toNode, string edgeData) {
433 
434     Edge* edge = new Edge(this, toNode, edgeData);
435     edges.push_back(edge);
436 
437     return edge;
438 }
439 
440 Edge* Node::AddEdgeFrom(Node* fromNode, string edgeData) {
441 
442     Edge* edge = new Edge(fromNode, this, edgeData);
443     fromNode->edges.push_back(edge);
444 
445     return edge;
446 }
447 
448 Edge::Edge(Node* source, Node* target) {
449 
450     this->source = source;
451     this->target = target;
452 
453 }
454 
455 Edge::Edge(Node* source, Node* target, string data) {
456 
457     this->data = data;
458     this->source = source;
459     this->target = target;
460 
461 }
462 
463 GenericObject* Edge::Output() {
464     GenericObject* ob = new GenericObject();
465 
466     ob->Add("data", CreateLeaf(data));
467     ob->Add("source", CreateLeaf(source->id));
468     ob->Add("target", CreateLeaf(target->id));
```

```
469
470     return ob;
471 }
472
473 void Edge::Print() {
474     cout << "Edge from " << source->id << " to " << target->id;
475     if (data != "")
476         cout << " (" << data << ")";
477     cout << endl;
478 }
```