```
 1 (s0, {) -> s1
 2 (s0, }) -> s2
 3 (s0, () -> s3
 4 (s0, )) -> s4
 5 (s0, [) -> s5
 6 (s0, ]) -> s6
 7 (s0, *) -> s7
 8 (s0, %) -> s8
 9 (s0, -) -> s9
10 (s0, +) -> s11
11 (s0, =) -> s13
12 (s0, <) -> s15
13 (s0, >) -> s17
14 (s0, !) -> s19
15 (s0, &) -> s21
16 (s0, |) -> s23
17 (s0, ,) -> s25
18 (s0, .) -> s26
19 (s0, ;) -> s31
20 (s0, /) -> s32
21 (s0, :) -> s40
22 (s0, #) -> s42
23 (s0, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
24 (s0, 1234567890) -> s30
25 (s0, """") -> s33
26 (s7, =) -> s45
27 (s9, -) -> s10
28 (s9, =) -> s44
29 (s9, >) -> s39
30 (s9, .) -> s27
31 (s9, 1234567890) -> s30
32 (s11, +) -> s12
33 (s11, =) -> s43
34 (s11, .) -> s27
35 (s11, 1234567890) -> s30
36 (s13, =) -> s14
37 (s15, =) -> s16
38 (s17, =) -> s18
39 (s19, =) -> s20
40 (s21, &) -> s22
41 (s23, |) -> s24
42 (s26, 1234567890) -> s27
43 (s27, 1234567890) -> s27
44 (s28, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
45 (s28, 1234567890) -> s29
46 (s29, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
47 (s29, 1234567890) -> s29
48 (s30, .) -> s27
49 (s30, 1234567890) -> s30
50 (s32, *) -> s36
51 (s32, =) -> s46
52 (s32, /) -> s35
53 (s33, {}()[]*%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890\t @$'?\\\n\r) -> s33
54 (s33, """") -> s34
55 (s35, {}()[]*%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\) -> s35
56 (s36, {}()[]%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\\n\r) -> s36
57 (s36, *) -> s37
58 (s37, {}()[]*%-+=<>!&|,.;:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\\n\r) -> s36
59 (s37, /) -> s38
60 (s40, :) -> s41
61
62 I(s0)
63 F(s1, L_CU_BRACKET)
64 F(s2, R_CU_BRACKET)
65 F(s3, L_PAREN)
66 F(s4, R_PAREN)
```

```
 67 F(s5, L_SQ_BRACKET)
 68 F(s6, R_SQ_BRACKET)
 69 F(s7, ASTERISK)
 70 F(s8, MOD)
 71 F(s9, MINUS)
 72 F(s10, DECR)
 73 F(s11, PLUS)
 74 F(s12, INCR)
 75 F(s13, ASSIGN)
 76 F(s14, EQ)
 77 F(s15, LT)
 78 F(s16, LT_EQ)
 79 F(s17, GT)
 80 F(s18, GT_EQ)
 81 F(s19, NOT)
 82 F(s20, NOT_EQ)
 83 F(s21, BITAND)
 84 F(s22, AND)
 85 F(s23, BITOR)
 86 F(s24, OR)
 87 F(s25, COMMA)
 88 F(s26, PERIOD)
 89 F(s27, FLOAT_LITERAL)
 90 F(s28, ID)
 91 F(s29, ID)
 92 F(s30, INT_LITERAL)
 93 F(s31, SEMICOLON)
 94 F(s32, SLASH)
 95 F(s34, STRING_LITERAL)
 96 F(s35, LINE_COMMENT)
 97 F(s38, BLOCK_COMMENT)
 98 F(s39, ARROW)
 99 F(s41, SCOPE_RESOLUTION)
100 F(s42, POUND)
101 F(s43, PLUS_ASSIGN)
102 F(s44, MINUS_ASSIGN)
103 F(s45, ASTERISK_ASSIGN)
104 F(s46, SLASH_ASSIGN)
105
106 T(L_CU_BRACKET, "{")
107 T(R_CU_BRACKET, "}")
108 T(L_PAREN, "(")
109 T(R_PAREN, ")")
110 T(L_SQ_BRACKET, "[")
111 T(L_SQ_BRACKET, "]")
112 T(ASTERISK, "*")
113 T(MOD, "%")
114 T(MINUS, "-")
115 T(DECR, "--")
116 T(PLUS, "+")
117 T(INCR, "++")
118 T(ASSIGN, "=")
119 T(EQ, "==")
120 T(LT, "<")
121 T(LT_EQ, "<=")
122 T(GT, ">")
123 T(GT_EQ, ">=")
124 T(NOT, "!")
125 T(NOT_EQ, "!=")
126 T(BITAND, "&")
127 T(AND, "&&")
128 T(BITOR, "|")
129 T(OR, "||")
130 T(COMMA, ",")
131 T(PERIOD, ".")
132 T(SEMICOLON, ";")
133 T(SLASH, "/")
```

```
134
135 ReservedWord(int, INT)
136 ReservedWord(integer, INTEGER)
137 ReservedWord(long, LONG)
138 ReservedWord(short, SHORT)
139 ReservedWord(byte, BYTE)
140 ReservedWord(float, FLOAT)
141 ReservedWord(double, DOUBLE)
142 ReservedWord(real, REAL)
143 ReservedWord(precision, PRECISION)
144 ReservedWord(fixed, FIXED)
145 ReservedWord(char, CHAR)
146 ReservedWord(character, CHARACTER)
147 ReservedWord(bool, BOOL)
148 ReservedWord(boolean, BOOLEAN)
149 ReservedWord(void, VOID)
150 ReservedWord(true, TRUE)
151 ReservedWord(false, FALSE)
152 ReservedWord(for, FOR)
153 ReservedWord(while, WHILE)
154 ReservedWord(do, DO)
155 ReservedWord(if, IF)
156 ReservedWord(else, ELSE)
157 ReservedWord(return, RETURN)
158 ReservedWord(class, CLASS)
159 ReservedWord(using, USING)
160 ReservedWord(namespace, NAMESPACE)
161 ReservedWord(include, INCLUDE)
162
163 Ignore(LINE_COMMENT)
164 Ignore(BLOCK_COMMENT)
165
166
167 include =:
168 [POUND] [INCLUDE] [STRING_LITERAL]
169
170 using-namespace =:
171 [USING] [NAMESPACE] <namespace-identifier> [SEMICOLON]
172
173 function-prototype =:
174 <function-return-type> <function-identifier> <function-proto-parameters> [SEMICOLON];
175
176
177 function-definition =:
178 <function-return-type> <function-identifier> <function-parameters> <function-body>
179
180 statement-list =:
181 <statement> <statement-list>?
182
183 statement =:
184 <expression-statement>|<for-loop>|<while-loop>|<decision>|<block>
185
186 expression-statement =:
187 <expression> [SEMICOLON]
188
189 while-loop =:
190 [WHILE] [L_PAREN] <while-condition> [R_PAREN] <while-body>
191
192 for-loop =:
193 [FOR] [L_PAREN] <for-init> [SEMICOLON] <for-condition> [SEMICOLON] <for-increment> [R_PAREN] <for-body>
194
195 decision =:
196 [IF] [L_PAREN] <expression> [R_PAREN] <decision-body> <decision-cases>? <decision-fallback>?
197
198 decision-cases =:
199 <decision-case> <decision-cases>?
200
```

```
decision-case =:
[ELSE] [IF] [L_PAREN] <expression> [R_PAREN] <decision-body>

decision-fallback =:
[ELSE] <decision-body>

decision-body =:
<block>|<statement>|[SEMICOLON]

block =:
[L_CU_BRACKET] <statement-list>? [R_CU_BRACKET]

member-access =:
<member-access-head> <member-access-tail>

method-invocation =:
<function-identifier> [L_PAREN] <arg-list>? [R_PAREN]

expression =:
<grouped-expression>|<method-invocation>|<declaration>|<assignment>|<operation>|<return>|<simple-expression>

simple-expression =:
<member-access>|<subscript-access>|<literal>|<identifier>

operation =:
<binary-expression>|<unary-expression>

subscript-access =:
<subscript-access-head> <object-subscript>

subscript-access-head =:
<method-invocation>|<identifier>

grouped-expression =:
[L_PAREN] <expression> [R_PAREN]

declaration =:
<type> <initializer-list>

initializer-list =:
<identifier> $DeclareVar([0], ("declaration")[0])$ <initializer-assignment-tail>? <initializer-list-tail>?

initializer-list-tail =:
[COMMA] <identifier> $DeclareVar([1], ("declaration")[0])$ <initializer-assignment-tail>? <initializer-list-tail>?

assignment =:
<assignment-target> <assignment-tail>

assignment-target =:
<member-access>|<setter>

binary-expression =:
<relational-expression>|<algebraic-expression>|<logical-expression>

relational-expression =:
<operation-expression> <relational-expression-tail>

algebraic-expression =:
<operation-expression> <algebraic-expression-tail>

logical-expression =:
<operation-expression> <logical-expression-tail>

operation-expression =:
<grouped-expression>|<literal>|<identifier>


```

```
268 relational-expression-tail =:
269 <relational-binary-op> <operation-expression> <relational-expression-tail>?
270
271 algebraic-expression-tail =:
272 <math-binary-op> <operation-expression> <algebraic-expression-tail>?
273
274 logical-expression-tail =:
275 <logical-binary-op> <operation-expression> <logical-expression-tail>?
276
277
278 unary-expression =:
279 <not-expression>|<unary-postfix-expression>|<unary-prefix-expression>
280
281 not-expression =:
282 [NOT] <identifier>
283
284 unary-postfix-expression =:
285 <identifier> <unary-op> $AccumulateVar([1], [0])$
286
287 unary-prefix-expression =:
288 <unary-op> <identifier> $AccumulateVar([0], [1])$
289
290 literal =:
291 <bool-literal>|[FLOAT_LITERAL]|[INT_LITERAL]|[STRING_LITERAL]
292
293 type =:
294 <primitive-type>|<identifier> <asterisk-tail>?
295
296 asterisk-tail =:
297 [ASTERISK] <asterisk-tail>?
298
299 primitive-type =:
300 <int-primitive>|<float-primitive>|<fixed-primitive>|<char-primitive>|<bool-primitive>|<void>
301
302 binary-op =:
303 <math-binary-op>|<logical-binary-op>|<relational-binary-op>
304
305 return =:
306 [RETURN] <expression>?
307
308 identifier =:
309 [ID]
310
311
312 void =:
313 [VOID]
314
315 int-primitive =:
316 [INT]|[LONG]|[SHORT]|[BYTE]|[INTEGER]
317
318 float-primitive =:
319 [FLOAT]|[DOUBLE]|[REAL]|<double-precision>
320
321 double-precision =:
322 [DOUBLE] [PRECISION]
323
324 fixed-primitive =:
325 [FIXED]
326
327 char-primitive =:
328 [CHAR]|[CHARACTER]
329
330 bool-primitive =:
331 [BOOL]|[BOOLEAN]
332
333 bool-literal =:
334 [TRUE]|[FALSE]
```

```
335
336 math-assign-op =:
337 [PLUS_ASSIGN]|[MINUS_ASSIGN]|[ASTERISK_ASSIGN]|[SLASH_ASSIGN]
338
339 math-binary-op =:
340 [PLUS]|[MINUS]|[ASTERISK]|[SLASH]
341
342 logical-binary-op =:
343 [BITAND]|[AND]|[BITOR]|[OR]
344
345 relational-binary-op =:
346 [EQ]|[LT]|[LT_EQ]|[GT]|[GT_EQ]|[NOT_EQ]
347
348 unary-op =:
349 [INCR]|[DECR]
350
351
352 declaration-list =:
353 <declaration> <declaration-list-tail>?
354
355 arg-list =:
356 <expression> <arg-list-tail>?
357
358 declaration-list-tail =:
359 [COMMA] <declaration> <declaration-list-tail>?
360
361 arg-list-tail =:
362 [COMMA] <expression> <arg-list-tail>?
363
364 member-access-head =:
365 <getter>
366
367 member-access-operator =:
368 [PERIOD]|[ARROW]
369
370 member-access-tail =:
371 <member-access-operator> <getter> <member-access-tail>?
372
373
374 object-subscript =:
375 [L_SQ_BRACKET] <expression> [R_SQ_BRACKET]
376
377 function-return-type =:
378 <type>
379
380 function-identifier =:
381 [ID]
382
383 function-proto-parameters =:
384 [L_PAREN] <function-proto-parameter-list>? [R_PAREN]
385
386 function-parameters =:
387 [L_PAREN] <function-parameter-list>? [R_PAREN]
388
389 function-parameter-list =:
390 <function-parameter-declaration> <function-parameter-list-tail>?
391
392 function-parameter-list-tail =:
393 [COMMA] <function-parameter-declaration> <function-parameter-list-tail>?
394
395 function-proto-parameter-list =:
396 <function-proto-parameter-declaration> <function-proto-parameter-list-tail>?
397
398 function-proto-parameter-list-tail =:
399 [COMMA] <function-proto-parameter-declaration> <function-proto-parameter-list-tail>?
400
401 function-body =:
```

```
<block>


function-parameter-declaration =:
<type> <identifier> $DeclareVar([1], [0])$ <function-parameter-assignment>?

function-proto-parameter-declaration =:
<type> <identifier>? <function-parameter-assignment>?

function-parameter-assignment =:
[ASSIGN] <assign-expression>

for-init =:
<expression>?

for-condition =:
<expression>?

for-increment =:
<expression>?

for-body =:
<block>|<statement>|[SEMICOLON]

while-condition =:
<expression>?

while-body =:
<block>|<statement>|[SEMICOLON]

assignment-tail =:
<algebraic-assignment-tail>|<standard-assignment-tail>

initializer-assignment-tail =:
[ASSIGN] <assign-expression> $AssignVar(^[@-1], ResolveExpr([1]))$

standard-assignment-tail =:
[ASSIGN] <assign-expression> $AssignVar(^^[@-1], ResolveExpr([1]))$

algebraic-assignment-tail =:
<math-assign-op> <assign-expression> $AccumulateVar([0][0], ^^[@-1], [1])$

assign-expression =:
<grouped-expression>|<method-invocation>|<assignment>|<operation>|<simple-expression>

namespace-identifier =:
[ID] <namespace-identifer-tail>?

namespace-identifier-tail =:
[SCOPE_RESOLUTION] [ID] <namespace-identifier-tail>?


getter =:
<method-invocation>|<subscript-access>|<identifier>

setter =:
<subscript-access>|<identifier>
```

```
 1 (s0, {) -> s1
 2 (s0, }) -> s2
 3 (s0, () -> s3
 4 (s0, )) -> s4
 5 (s0, [) -> s5
 6 (s0, ]) -> s6
 7 (s0, *) -> s7
 8 (s0, %) -> s8
 9 (s0, -) -> s9
10 (s0, +) -> s11
11 (s0, =) -> s13
12 (s0, <) -> s15
13 (s0, >) -> s17
14 (s0, !) -> s19
15 (s0, &) -> s21
16 (s0, |) -> s23
17 (s0, ,) -> s25
18 (s0, .) -> s26
19 (s0, ;) -> s31
20 (s0, /) -> s32
21 (s0, $_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
22 (s0, 1234567890) -> s30
23 (s0, """") -> s33
24 (s7, =) -> s41
25 (s9, -) -> s10
26 (s9, =) -> s40
27 (s9, .) -> s27
28 (s9, 1234567890) -> s30
29 (s11, +) -> s12
30 (s11, =) -> s39
31 (s11, .) -> s27
32 (s11, 1234567890) -> s30
33 (s13, =) -> s14
34 (s15, =) -> s16
35 (s17, =) -> s18
36 (s19, =) -> s20
37 (s21, &) -> s22
38 (s23, |) -> s24
39 (s26, 1234567890) -> s27
40 (s27, 1234567890) -> s27
41 (s28, $_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
42 (s28, 1234567890) -> s29
43 (s29, $_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
44 (s29, 1234567890) -> s29
45 (s30, .) -> s27
46 (s30, 1234567890) -> s30
47 (s32, *) -> s36
48 (s32, =) -> s42
49 (s32, /) -> s35
50 (s33, {}()[]*%-+=<>!&|,.;/$_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890\t @#$:'?\\\n\r) -> s33
51 (s33, """") -> s34
52 (s35, {}()[]*%-+=<>!&|,.;/$_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @#$:'?\\) -> s35
53 (s36, {}()[]%-+=<>!&|,.;/$_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @#$:'?\\\n\r) -> s36
54 (s36, *) -> s37
55 (s37, {}()[]*%-+=<>!&|,.;$_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @#$:'?\\\n\r) -> s36
56 (s37, /) -> s38
57
58 I(s0)
59 F(s1, L_CU_BRACKET)
60 F(s2, R_CU_BRACKET)
61 F(s3, L_PAREN)
62 F(s4, R_PAREN)
63 F(s5, L_SQ_BRACKET)
64 F(s6, R_SQ_BRACKET)
65 F(s7, ASTERISK)
66 F(s8, MOD)
```

```
 67 F(s9, MINUS)
 68 F(s10, DECR)
 69 F(s11, PLUS)
 70 F(s12, INCR)
 71 F(s13, ASSIGN)
 72 F(s14, EQ)
 73 F(s15, LT)
 74 F(s16, LT_EQ)
 75 F(s17, GT)
 76 F(s18, GT_EQ)
 77 F(s19, NOT)
 78 F(s20, NOT_EQ)
 79 F(s21, BITAND)
 80 F(s22, AND)
 81 F(s23, BITOR)
 82 F(s24, OR)
 83 F(s25, COMMA)
 84 F(s26, PERIOD)
 85 F(s27, FLOAT_LITERAL)
 86 F(s28, ID)
 87 F(s29, ID)
 88 F(s30, INT_LITERAL)
 89 F(s31, SEMICOLON)
 90 F(s32, SLASH)
 91 F(s34, STRING_LITERAL)
 92 F(s35, LINE_COMMENT)
 93 F(s38, BLOCK_COMMENT)
 94 F(s39, PLUS_ASSIGN)
 95 F(s40, MINUS_ASSIGN)
 96 F(s41, ASTERISK_ASSIGN)
 97 F(s42, SLASH_ASSIGN)
 98
 99 T(L_CU_BRACKET, "{")
100 T(R_CU_BRACKET, "}")
101 T(L_PAREN, "(")
102 T(R_PAREN, ")")
103 T(L_SQ_BRACKET, "[")
104 T(L_SQ_BRACKET, "]")
105 T(ASTERISK, "*")
106 T(MOD, "%")
107 T(MINUS, "-")
108 T(DECR, "--")
109 T(PLUS, "+")
110 T(INCR, "++")
111 T(ASSIGN, "=")
112 T(EQ, "==")
113 T(LT, "<")
114 T(LT_EQ, "<=")
115 T(GT, ">")
116 T(GT_EQ, ">=")
117 T(NOT, "!")
118 T(NOT_EQ, "!=")
119 T(BITAND, "&")
120 T(AND, "&&")
121 T(BITOR, "|")
122 T(OR, "||")
123 T(COMMA, ",")
124 T(PERIOD, ".")
125 T(SEMICOLON, ";")
126 T(SLASH, "/")
127
128 ReservedWord(var, VAR)
129 ReservedWord(true, TRUE)
130 ReservedWord(false, FALSE)
131 ReservedWord(for, FOR)
132 ReservedWord(while, WHILE)
133 ReservedWord(do, DO)
```

```
ReservedWord(if, IF)
ReservedWord(else, ELSE)
ReservedWord(return, RETURN)
ReservedWord(function, FUNCTION)

Ignore(LINE_COMMENT)
Ignore(BLOCK_COMMENT)


function-definition =:
[FUNCTION] <function-identifier> <function-parameters> <function-body>

statement-list =:
<statement> <statement-list>?

statement =:
<expression-statement>|<for-loop>|<while-loop>|<decision>|<block>

expression-statement =:
<expression> [SEMICOLON]

while-loop =:
[WHILE] [L_PAREN] <while-condition> [R_PAREN] <while-body>

for-loop =:
[FOR] [L_PAREN] <for-init> [SEMICOLON] <for-condition> [SEMICOLON] <for-increment> [R_PAREN] <for-body>

decision =:
[IF] [L_PAREN] <expression> [R_PAREN] <decision-body> <decision-cases>? <decision-fallback>?

decision-cases =:
<decision-case> <decision-cases>?

decision-case =:
[ELSE] [IF] [L_PAREN] <expression> [R_PAREN] <decision-body>

decision-fallback =:
[ELSE] <decision-body>

decision-body =:
<block>|<statement>|[SEMICOLON]

block =:
[L_CU_BRACKET] <statement-list>? [R_CU_BRACKET]

member-access =:
<member-access-head> <member-access-tail>

method-invocation =:
<function-identifier> [L_PAREN] <arg-list>? [R_PAREN]

expression =:
<grouped-expression>|<method-invocation>|<declaration>|<assignment>|<operation>|<return>|<simple-expression>

simple-expression =:
<member-access>|<subscript-access>|<literal>|<identifier>

operation =:
<binary-expression>|<unary-expression>

subscript-access =:
<subscript-access-head> <object-subscript>

subscript-access-head =:
<member-access>|<method-invocation>|<identifier>

grouped-expression =:
```

```
[L_PAREN] <expression> [R_PAREN]

declaration =:
[VAR] <initializer-list>

initializer-list =:
<identifier> <initializer-assignment-tail>? <initializer-list-tail>?

initializer-list-tail =:
[COMMA] <identifier> <initializer-assignment-tail>? <initializer-list-tail>?

assignment =:
<assignment-target> <assignment-tail>

assignment-target =:
<member-access>|<setter>

binary-expression =:
<relational-expression>|<algebraic-expression>|<logical-expression>

relational-expression =:
<operation-expression> <relational-expression-tail>

algebraic-expression =:
<operation-expression> <algebraic-expression-tail>

logical-expression =:
<operation-expression> <logical-expression-tail>

operation-expression =:
<grouped-expression>|<literal>|<identifier>


relational-expression-tail =:
<relational-binary-op> <operation-expression> <relational-expression-tail>?

algebraic-expression-tail =:
<math-binary-op> <operation-expression> <algebraic-expression-tail>?

logical-expression-tail =:
<logical-binary-op> <operation-expression> <logical-expression-tail>?


unary-expression =:
<not-expression>|<unary-postfix-expression>|<unary-prefix-expression>

not-expression =:
[NOT] <identifier>

unary-postfix-expression =:
<identifier> <unary-op>

unary-prefix-expression =:
<unary-op> <identifier>

literal =:
<bool-literal>|[FLOAT_LITERAL]|[INT_LITERAL]|[STRING_LITERAL]

binary-op =:
<math-binary-op>|<logical-binary-op>|<relational-binary-op>

return =:
[RETURN] <expression>?

identifier =:
[ID]

```

```
268
269 bool-literal =:
270 [TRUE]|[FALSE]
271
272 math-binary-op =:
273 [PLUS]|[MINUS]|[ASTERISK]|[SLASH]
274
275 logical-binary-op =:
276 [BITAND]|[AND]|[BITOR]|[OR]
277
278 relational-binary-op =:
279 [EQ]|[LT]|[LT_EQ]|[GT]|[GT_EQ]|[NOT_EQ]
280
281 unary-op =:
282 [INCR]|[DECR]
283
284
285 declaration-list =:
286 <declaration> <declaration-list-tail>?
287
288 arg-list =:
289 <expression> <arg-list-tail>?
290
291 declaration-list-tail =:
292 [COMMA] <declaration> <declaration-list-tail>?
293
294 arg-list-tail =:
295 [COMMA] <expression> <arg-list-tail>?
296
297 member-access-head =:
298 <getter>
299
300 member-access-operator =:
301 [PERIOD]
302
303 member-access-tail =:
304 <member-access-operator> <getter> <member-access-tail>?
305
306
307 object-subscript =:
308 [L_SQ_BRACKET] <expression> [R_SQ_BRACKET]
309
310 function-identifier =:
311 [ID]
312
313 function-parameters =:
314 [L_PAREN] <function-parameter-list>? [R_PAREN]
315
316 function-parameter-list =:
317 <function-parameter-declaration> <function-parameter-list-tail>?
318
319 function-parameter-list-tail =:
320 [COMMA] <function-parameter-declaration> <function-parameter-list-tail>?
321
322 function-body =:
323 <block>
324
325 function-parameter-declaration =:
326 <identifier>
327
328 for-init =:
329 <expression>?
330
331 for-condition =:
332 <expression>?
333
334 for-increment =:
```

```
335 <expression>?
336
337 for-body =:
338 <block>|<statement>|[SEMICOLON]
339
340 while-condition =:
341 <expression>?
342
343 while-body =:
344 <block>|<statement>|[SEMICOLON]
345
346 assignment-tail =:
347 <algebraic-assignment-tail>|<standard-assignment-tail>
348
349 initializer-assignment-tail =:
350 [ASSIGN] <assign-expression>
351
352 standard-assignment-tail =:
353 [ASSIGN] <assign-expression>
354
355 algebraic-assignment-tail =:
356 <math-assign-op> [ASSIGN] <assign-expression>
357
358 math-assign-op =:
359 [PLUS_ASSIGN]|[MINUS_ASSIGN]|[ASTERISK_ASSIGN]|[SLASH_ASSIGN]
360
361 assign-expression =:
362 <grouped-expression>|<method-invocation>|<assignment>|<operation>|<return>|<simple-expression>
363
364 getter =:
365 <method-invocation>|<subscript-access>|<identifier>
366
367 setter =:
368 <subscript-access>|<identifier>
```

```cpp
#include "AnalyzeModule.h"

static string _AnalyzeModule = RegisterPlugin("Analyze", new AnalyzeModule());

AnalyzeModule::AnalyzeModule() {}

CASP_Return* AnalyzeModule::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 " << _AnalyzeModule << " Module!\n";

    GetAllAnalyses(markup);

    for (auto it = functionTable.begin(); it != functionTable.end(); it++) {
        bool undefined = it->second == NULL || it->second->IsUndefined();
        string analysis = it->second != NULL ? it->second->ToString() : "Undefined";
        GenericObject* ob = CreateObject({
            { "IsUndefined", CreateLeaf(undefined) },
            { "Analysis", CreateLeaf(analysis) },
            { "Title", CreateLeaf(it->first) }
        });
        returnData->Data()->Add(it->first, ob);
        // if (it->second != NULL) {
        //     cout << it->first << ": O(" << it->second->ToString() << ")" << endl;
        // } else {
        //     cout << it->first << ": Undefined" << endl;
        // }
    }

    return returnData;
}

void AnalyzeModule::GetAllAnalyses(Markup* masterTree) {
    vector<Markup*> functions = masterTree->FindAllById("function-definition", true);
    vector<Markup*> sls = masterTree->FindAllChildrenById("statement-list");

    if (sls.size() > 0) {
        GetRootAnalysis(sls);
    }
    if (functions.size() > 0) {
        int i;
        for (i = 0; i < functions.size(); i++) {
            string fnName = functions[i]->FindFirstChildById("function-identifier")->GetData();
            markupTable[fnName] = functions[i];
        }
        for (i = 0; i < functions.size(); i++) {
            GetFunctionAnalysis(functions[i]);
        }
    }

}

Analysis* AnalyzeModule::GetRootAnalysis(vector<Markup*> parseTrees) {

    AnalysisTree* analysis = new AnalysisTree();

    for (int i = 0; i < parseTrees.size(); i++) {
        processBlock(parseTrees[i], analysis);
    }

    return functionTable["ROOT"] = analysis->GetAnalysis();
```

```
 67 }
 68
 69 Analysis* AnalyzeModule::GetFunctionAnalysis(Markup* functionTree) {
 70
 71     if (functionTree == NULL)
 72         return NULL;
 73
 74     string functionTitle = functionTree->FindFirstChildById("function-identifier")->GetData();
 75
 76     if (functionTable[functionTitle] == NULL) {
 77         AnalysisTree* analysis = new AnalysisTree();
 78         Markup* block = functionTree->FindFirstById("block");
 79         processBlock(block, analysis);
 80         functionTable[functionTitle] = analysis->GetAnalysis();
 81     }
 82
 83     return functionTable[functionTitle];
 84 }
 85
 86 void AnalyzeModule::analyzeMethodCall(Markup* parseTree, AnalysisTree* analysis) {
 87     string functionTitle = parseTree->FindFirstChildById("function-identifier")->GetData();
 88
 89     Analysis* fnAnalysis = GetFunctionAnalysis(markupTable[functionTitle]);
 90
 91     // todo - Add a warning if the function doesn't exist
 92     AnalysisTree* node = new AnalysisTree();
 93     node->SetAnalysis(fnAnalysis);
 94
 95     analysis->AddChild(node);
 96 }
 97
 98 void AnalyzeModule::analyzeDecision(Markup* parseTree, AnalysisTree* analysis) {
 99
100     AnalysisTree* node = new AnalysisTree();
101     analysis->AddChild(node);
102     // analyze each block, add worst-case block to analysis
103
104     /* each block in tree stored as a list nlogn would be push as n,logn
105     */
106
107     Markup* condition = parseTree->FindFirstChildById("expression");
108     Markup* body = parseTree->FindFirstChildById("decision-body");
109     Markup* proc;
110     vector<Markup*> decisionCases = parseTree->FindFirstChildById("decision-cases")->RecursiveElements();
111     Markup* fallback = parseTree->FindFirstChildById("decision-fallback");
112
113     // process if expression here, too
114     if ((proc = body->FindFirstChildById("block")) != NULL) {
115         processBlock(proc, node);
116     }
117     else if ((proc = body->FindFirstChildById("statement")) != NULL) {
118         processStatement(proc, node);
119     }
120
121     for (int i = 0; i < decisionCases.size(); i++) {
122         // create a new tree node and append it to the current tree?
123         // process else-if expression here, too
124         Markup* dc = decisionCases[i]->FindFirstChildById("decision-case");
125         condition = dc->FindFirstChildById("expression");
126         body = dc->FindFirstChildById("decision-body");
127
128         if ((proc = body->FindFirstChildById("block")) != NULL) {
129             processBlock(proc, node);
130         }
131         else if ((proc = body->FindFirstChildById("statement")) != NULL) {
132             processStatement(proc, node);
133         }
```

```
134     }
135
136     if (fallback != NULL) {
137         // create a new tree node and append it to the current tree?
138         body = fallback->FindFirstChildById("decision-body");
139         if ((proc = body->FindFirstChildById("block")) != NULL) {
140             processBlock(proc, node);
141         }
142         else if ((proc = body->FindFirstChildById("statement")) != NULL) {
143             processStatement(proc, node);
144         }
145     }
146
147 }
148
149 void AnalyzeModule::analyzeProcess(Markup* parseTree, AnalysisTree* analysis) {
150
151     AnalysisTree* tree = new AnalysisTree();
152     tree->AddConstantFactor();
153
154     analysis->AddChild(tree);
155
156 }
157 void AnalyzeModule::analyzeLoop(Markup* parseTree, AnalysisTree* analysis) {
158
159     Markup* init = parseTree->FindFirstChildById("for-init")->ChildAt(0);
160     Markup* condition = parseTree->FindFirstChildById("for-condition")->ChildAt(0);
161     Markup* increment = parseTree->FindFirstChildById("for-increment")->ChildAt(0);
162     Markup* body = parseTree->FindFirstChildById("for-body");
163     Markup* proc = NULL;
164
165     AnalysisTree* tree = new AnalysisTree();
166     analysis->AddChild(tree);
167
168     AnalysisNode* a = new AnalysisNode();
169     a->SetToUndefined();
170
171     string conditionalOp = "";
172     int incrVal = 0;
173     string id = "";
174     string conditional = "";
175     bool idValSet = false;
176     int idVal = 0;
177     bool conditionalValSet = false;
178     int conditionalVal = 0;
179
180     unordered_map<string, Markup*> declaredIds;
181
182
183     if (init != NULL || condition != NULL || increment != NULL) {
184         // TODO if the incremented id is declared outside of the for loop, this won't operate correctly
185         if (init != NULL) {
186             // Get initial condition
187             Markup* assign = NULL;
188
189             if ((assign = init->FindFirstChildById("assignment")) != NULL) {
190                 string ident = assign->FindFirstChildById("assignment-target")->GetData();
191                 Markup* expr = assign->FindFirstChildById("assignment-tail")->ChildAt(0)->FindFirstChildById("assign-expression")->ChildAt(0);
192
193                 declaredIds[ident] = ActionRoutines::ExecuteAction("ResolveExpr", parseTree, { expr });
194
195             } else if ((assign = init->FindFirstChildById("declaration")) == NULL) {
196
197                 Markup* start = init->FindFirstById("initializer-list");
198                 vector<Markup*> recursive = start->RecursiveElements();
199                 vector<Markup*> list = { start };
200                 list.insert(list.end(), recursive.begin(), recursive.end());
```

```
201
202             for (int i = 0; i < list.size(); i++) {
203                 string ident = list[i]->FindFirstChildById("identifier")->GetData();
204                 Markup* expr = list[i]->FindFirstChildById("initializer-assignment-tail")->ChildAt(0)->FindFirstChildById("assign-expression")->ChildAt(0);
205             cout << "Declared " << ident << endl;
206
207                 declaredIds[ident] = ActionRoutines::ExecuteAction("ResolveExpr", parseTree, { expr });
208             }
209
210         } else {
211             // there is no definition here, look for it elsewhere based on the condition/increment?
212         }
213
214     }
215     if(increment != NULL){
216         // get increment
217         Markup* operation;
218         if ((operation = increment->FindFirstChildById("assignment")) != NULL) {
219             id = operation->FindFirstChildById("assignment-target")->GetData();
220             if (declaredIds[id] != NULL && declaredIds[id]->GetID() == "INT_LITERAL") {
221                 idValSet = true;
222                 idVal = stoi(declaredIds[id]->GetData());
223             }
224             Markup* tail = operation->FindFirstChildById("assignment-tail");
225             Markup* t = NULL;
226             if ((t = tail->FindFirstChildById("algebraic-assignment-tail")) != NULL) {
227                 Markup* op = t->FindFirstChildById("math-assign-op")->ChildAt(0);
228                 Markup* expr = ActionRoutines::ExecuteAction("ResolveExpr", parseTree, { t->FindFirstChildById("assign-expression")->ChildAt(0) });
229
230                 if (expr->GetID() == "INT_LITERAL") {
231                     string opId = op->GetID();
232                     if (opId == "PLUS_ASSIGN" || opId == "MINUS_ASSIGN") {
233                         a->SetToExponential(1);
234                     } else if (opId == "ASTERISK_ASSIGN" || opId == "SLASH_ASSIGN") {
235                         int base = 10; //stoi(expr->GetData());
236                         a->SetToLogarithmic(base, 1);
237                     }
238                     incrVal = stoi(expr->GetData());
239                     conditionalOp = opId;
240                 } else {
241                     // unable to calculate
242                 }
243
244             } else if ((t = tail->FindFirstChildById("standard-assignment-tail")) != NULL) {
245                 // TODO This is potentially complex logic
246             }
247         }
248         else if ((operation = increment->FindFirstChildById("operation")) != NULL) {
249             Markup* unary = operation->FindFirstChildById("unary-expression");
250             if (unary != NULL) {
251                 string opType = unary->ChildAt(0)->GetID();
252                 if (opType == "unary-postfix-expression" || opType == "unary-prefix-expression") {
253                     Markup* op = unary->ChildAt(0)->FindFirstChildById("unary-op");
254                     Markup* identifier = unary->ChildAt(0)->FindFirstChildById("identifier");
255                     id = identifier->GetData();
256                     opType = op->ChildAt(0)->GetID();
257                     if(opType == "INCR") {
258                         a->SetToExponential(1);
259                         conditionalOp = "PLUS";
260                     } else if(opType == "DECR") {
261                         a->SetToExponential(1);
262                         conditionalOp = "MINUS";
263                     }
264                     incrVal = 1;
265                 }
266             } else {
267                 // any other operation does nothing
```

```
268                 }
269             } else {
270                 // can't get an increment
271             }
272         }
273         if(condition != NULL){
274             // Get final condition
275             Markup* operation = condition->ChildAt(0)->ChildAt(0)->FindFirstChildById("relational-expression");
276             if (operation != NULL) {
277                 Markup* lExpr = operation->FindFirstChildById("operation-expression")->ChildAt(0);
278                 lExpr = ActionRoutines::ExecuteAction("ResolveExpr", parseTree, { lExpr });
279                 Markup* rExpr = operation->FindFirstChildById("relational-expression-tail")->FindFirstChildById("operation-expression")->ChildAt(0);
280                 rExpr = ActionRoutines::ExecuteAction("ResolveExpr", parseTree, { rExpr });
281                 Markup* op = operation->FindFirstChildById("relational-expression-tail")->FindFirstChildById("relational-binary-op")->ChildAt(0);
282                 string opType = op->GetID();
283                 // id = identifier->GetData();
284                 Markup* lit = operation->FindFirstById("INT_LITERAL"); // could be float literal or id?
285                 string lType = lExpr->GetID();
286                 string rType = rExpr->GetID();
287 
288                 // the calculation can only be done right now if at least one side resolves to an ID
289                 if (lType == "ID" || rType == "ID") {
290                     if (lExpr->GetData() == id) {
291                         if (rType == "INT_LITERAL") {
292                             conditionalValSet = true;
293                             conditionalVal = stoi(rExpr->GetData());
294                         }
295                         conditional = opType;
296                     } else if (rExpr->GetData() == id) {
297                         if (lType == "INT_LITERAL") {
298                             conditionalValSet = true;
299                             conditionalVal = stoi(lExpr->GetData());
300                         }
301                         // reverse the operator to move the conditional operand to the right side
302                         if (opType == "LT") {
303                             conditional = "GT";
304                         } else if (opType == "LT_EQ") {
305                             conditional = "GT_EQ";
306                         } else if (opType == "GT") {
307                             conditional = "LT";
308                         } else if (opType == "GT_EQ") {
309                             conditional = "LT_EQ";
310                         }
311                     }
312                 }
313 
314             } else {
315                 // there is no relational condition
316             }
317         }
318 
319         if (conditional != "" && conditionalOp != "") {
320             if (conditionalOp == "LT" || conditionalOp == "LT_EQ") {
321                 if (((conditionalOp == "MINUS" && incrVal >= 0) ||
322                     (conditionalOp == "PLUS" && incrVal <= 0) ||
323                     (conditionalOp == "ASTERISK" && incrVal <= 1 && incrVal > -1) ||
324                     (conditionalOp == "SLASH" && (incrVal >= 1 || incrVal <= -1)))) {
325                     // TODO add warning for probable infinite loop
326                     a->SetToUndefined();
327                 } else if (!(conditionalValSet && idValSet) && (conditionalValSet || idValSet)) {
328                     if (conditionalOp == "MINUS" || conditionalOp == "PLUS")
329                         a->SetToExponential(1);
330                     else if (conditionalOp == "SLASH" || conditionalOp == "ASTERISK")
331                         a->SetToLogarithmic(10/*incrVal*/, 1);
332                 }
333                 //
334             } else if (conditionalOp == "GT" || conditionalOp == "GT_EQ") {
```

```
335                 if (((conditionalOp == "MINUS" && incrVal <= 0) ||
336                     (conditionalOp == "PLUS" && incrVal >= 0) ||
337                     (conditionalOp == "ASTERISK" && (incrVal >= 1 || incrVal <= -1)) ||
338                     (conditionalOp == "SLASH" && incrVal <= 1 && incrVal > -1))) {
339
340                     // TODO add warning for probable infinite loop
341                     a->SetToUndefined();
342                 } else if (!(conditionalValSet && idValSet) && (conditionalValSet || idValSet)) {
343                     if (conditionalOp == "MINUS" || conditionalOp == "PLUS")
344                         a->SetToExponential(1);
345                     else if (conditionalOp == "SLASH" || conditionalOp == "ASTERISK")
346                         a->SetToLogarithmic(10/*incrVal*/, 1);
347                 }
348                 //
349             } else if (conditionalOp == "EQ") {
350                 // TODO requires extra calculation.
351             } else if (conditionalOp == "NOT_EQ") {
352                 // TODO requires extra calculation.
353             }
354         }
355
356     }
357
358     tree->AddFactor(a);
359
360     if ((proc = body->FindFirstChildById("block")) != NULL) {
361         processBlock(proc, tree);
362     } else if ((proc = body->FindFirstChildById("statement")) != NULL) {
363         processStatement(proc, tree);
364     }
365
366 }
367
368 void AnalyzeModule::processStatement(Markup* statement, AnalysisTree* analysis) {
369     Markup* s = statement->ChildAt(0);
370     string id = s->GetID();
371
372     if (id == "for-loop") {
373         analyzeLoop(s, analysis);
374     } else if (id == "decision") {
375         analyzeDecision(s, analysis);
376     } else if (id == "block") {
377         processBlock(s, analysis);
378     } else if (id == "expression-statement") {
379         s = s->ChildAt(0)->ChildAt(0);
380         id = s->GetID();
381         while (id == "grouped-expression") {
382             s = s->ChildAt(1);
383             id = s->GetID();
384         }
385
386         if (id == "method-invocation") {
387             analyzeMethodCall(s, analysis);
388         }
389         else {
390             analyzeProcess(s, analysis);
391         }
392     }
393 }
394 void AnalyzeModule::processBlock(Markup* parseTree, AnalysisTree* analysis) {
395     Markup* sl = parseTree->FindFirstById("statement-list");
396
397     Markup* cs = NULL;
398     int ct = 0;
399
400     while (sl != NULL) {
401         cs = sl->FindFirstChildById("statement");
```

```
402         processStatement(cs, analysis);
403         sl = sl->FindFirstChildById("statement-list");
404     }
405 }
406
407 AnalysisTree::AnalysisTree() {
408     analysis = new Analysis();
409     analysis->AddConstantFactor();
410 }
411
412 void AnalysisTree::AddChild(AnalysisTree* tree) {
413     children.push_back(tree);
414 }
415
416 void AnalysisTree::SetAnalysis(Analysis* analysis) {
417     this->analysis = analysis;
418 }
419
420 Analysis* AnalysisTree::GetAnalysis() {
421
422     if (children.size() > 0) {
423         Analysis* max = children[0]->GetAnalysis();
424         Analysis* c = NULL;
425         for (int i = 1; i < children.size(); i++) {
426             c = children[i]->GetAnalysis();
427             if (*c > *max)
428                 max = c;
429         }
430
431         return &(*analysis * *max);
432
433     } else {
434         return analysis;
435     }
436 }
437
438 void AnalysisTree::AddFactor(AnalysisNode* node) {
439     analysis->AddFactor(node);
440 }
441
442 void AnalysisTree::AddConstantFactor() {
443     analysis->AddConstantFactor();
444 }
445 void AnalysisTree::AddExponentialFactor(int exponent) {
446     analysis->AddExponentialFactor(exponent);
447 }
448 void AnalysisTree::AddLogarithmicFactor(int base, int exponent) {
449     analysis->AddLogarithmicFactor(base, exponent);
450 }
451
452 Analysis::Analysis() {
453
454 }
455
456 bool Analysis::IsUndefined() {
457     return undefined;
458 }
459
460 void Analysis::AddConstantFactor() {
461     AnalysisNode* node = new AnalysisNode();
462     node->SetToConstant();
463     AddFactor(node);
464 }
465 void Analysis::AddExponentialFactor(int exponent) {
466     AnalysisNode* node = new AnalysisNode();
467     node->SetToExponential(exponent);
468     AddFactor(node);
```

```
469 }
470 void Analysis::AddLogarithmicFactor(int base, int exponent) {
471     AnalysisNode* node = new AnalysisNode();
472     node->SetToLogarithmic(base, exponent);
473     AddFactor(node);
474 }
475
476 void Analysis::AddFactor(AnalysisNode* node) {
477     switch (node->type) {
478         case Undefined:
479             undefined = true;
480             break;
481         case Constant:
482             if (this->constant == NULL) {
483                 this->constant = node;
484             }
485             break;
486         case Exponential:
487             if (this->exponential == NULL) {
488                 this->exponential = node;
489             } else {
490                 this->exponential = &(*this->exponential * *node);
491             }
492             break;
493         case Logarithmic:
494             if (this->logarithmic == NULL) {
495                 this->logarithmic = node;
496             } else {
497                 this->logarithmic = &(*this->logarithmic * *node);
498             }
499             break;
500     }
501 }
502
503 string Analysis::ToString() {
504
505     string str = "";
506
507     if (undefined) {
508         str += "Undefined";
509     } else {
510
511         if (exponential != NULL) {
512             str += exponential->ToString();
513         }
514
515         if (logarithmic != NULL) {
516             if (str != "")
517                 str += " ";
518             str += logarithmic->ToString();
519         }
520
521         if (str == "" && constant != NULL) {
522             str += constant->ToString();
523         }
524
525     }
526
527     return str;
528 }
529
530 Analysis& Analysis::operator*(Analysis& r) {
531     Analysis* a = new Analysis();
532
533     if (r.undefined || this->undefined) {
534         a->undefined = true;
535         return *a;
```

```
536     }
537
538     if (this->exponential != NULL && r.exponential != NULL)
539         a->exponential = &(*(this->exponential) * *(r.exponential));
540     else if (this->exponential != NULL)
541         a->exponential = this->exponential;
542     else if (r.exponential != NULL)
543         a->exponential = r.exponential;
544
545     if (this->logarithmic != NULL && r.logarithmic != NULL)
546         a->logarithmic = &(*(this->logarithmic) * *(r.logarithmic));
547     else if (this->logarithmic != NULL)
548         a->logarithmic = this->logarithmic;
549     else if (r.logarithmic != NULL)
550         a->logarithmic = r.logarithmic;
551
552     if (this->constant != NULL)
553         a->constant = this->constant;
554     else if (r.constant != NULL)
555         a->constant = r.constant;
556
557     return *a;
558 }
559
560 bool operator==(const Analysis& l, const Analysis& r) {
561     bool same = true;
562
563     if (l.undefined && r.undefined)
564         return true;
565     else if (l.undefined || r.undefined)
566         return false;
567
568     if (l.exponential != NULL && r.exponential != NULL)
569         same = same && *(l.exponential) == *(r.exponential);
570     else if ((l.exponential == NULL || r.exponential == NULL) && !(l.exponential == NULL && r.exponential == NULL))
571         return false;
572
573     if (l.logarithmic != NULL && r.logarithmic != NULL)
574         same = same && *(l.logarithmic) == *(r.logarithmic);
575     else if ((l.logarithmic == NULL || r.logarithmic == NULL) && !(l.logarithmic == NULL && r.logarithmic == NULL))
576         return false;
577
578     if (l.exponential == NULL && r.exponential == NULL && l.logarithmic == NULL && r.logarithmic == NULL) {
579         if (l.constant != NULL && r.constant != NULL)
580             same = same && true;
581         else if ((l.constant == NULL || r.constant == NULL) && !(l.constant == NULL && r.constant == NULL))
582             return false;
583     }
584
585     return same;
586 }
587 bool operator!=(const Analysis& l, const Analysis& r) {
588     return !(l == r);
589 }
590 bool operator>(const Analysis& l, const Analysis& r) {
591     bool gtr = true;
592
593     if (l.undefined)
594         return false;
595     else if (r.undefined)
596         return true;
597
598     if (l.exponential != NULL && r.exponential != NULL) {
599         gtr = gtr && *(l.exponential) > *(r.exponential);
600     } else if (l.exponential == NULL && r.exponential == NULL) {
601
602     } else if (l.exponential == NULL) {
```

```
603         return false;
604     } else {
605         return true;
606     }
607 
608     if (l.logarithmic != NULL && r.logarithmic != NULL) {
609         gtr = gtr && *(l.logarithmic) > *(r.logarithmic);
610     } else if (l.logarithmic == NULL && r.logarithmic == NULL) {
611 
612     } else if (l.logarithmic == NULL) {
613         return false;
614     } else {
615         return true;
616     }
617 
618     if (l.exponential == NULL && r.exponential == NULL && l.logarithmic == NULL && r.logarithmic == NULL) {
619 
620         if (l.constant == NULL)
621             return false;
622         else if (r.constant == NULL)
623             return true;
624     }
625 
626     return gtr;
627 }
628 bool operator>=(const Analysis& l, const Analysis& r) {
629     return (l > r || l == r);
630 }
631 bool operator<(const Analysis& l, const Analysis& r) {
632     return (r > l);
633 }
634 bool operator<=(const Analysis& l, const Analysis& r) {
635     return (l < r || l == r);
636 }
637 
638 AnalysisNode::AnalysisNode() {}
639 
640 AnalysisNode& AnalysisNode::operator=(AnalysisNode& target) {
641     if (this != &target) {
642         this->type = target.type;
643         this->base = target.base;
644         this->exponent = target.exponent;
645     }
646     return *this;
647 }
648 AnalysisNode* AnalysisNode::operator=(AnalysisNode* target) {
649     if (this != target) {
650         this->type = target->type;
651         this->base = target->base;
652         this->exponent = target->exponent;
653     }
654     return this;
655 }
656 AnalysisNode& AnalysisNode::operator*(AnalysisNode& r) {
657     AnalysisNode* node = new AnalysisNode();
658 
659     if (this->type == r.type && (this->type != Logarithmic || this->base == r.base)) {
660         node = this;
661         node->exponent += r.exponent;
662     }
663 
664     return *node;
665 }
666 bool operator==(const AnalysisNode& l, const AnalysisNode& r) {
667     return (r.type == l.type && r.exponent == l.exponent && r.base == l.base);
668 }
669 bool operator!=(const AnalysisNode& l, const AnalysisNode& r) {
```

```
670     return !(l == r);
671 }
672 bool operator>(const AnalysisNode& l, const AnalysisNode& r) {
673     if (l.type != Undefined && r.type != Undefined) {
674 
675         if (l.type == r.type) {
676             if (l.exponent == r.exponent) {
677                 if (r.type == Logarithmic) {
678                     return !(r.base == l.base || r.base > l.base);
679                 } else {
680                     return false;
681                 }
682             } else {
683                 return (r.exponent < l.exponent);
684             }
685 
686         } else if (l.type == Constant || r.type == Exponential) {
687             return false;
688         } else if (l.type == Exponential || r.type == Constant) {
689             return true;
690         }
691 
692     } else if (l.type == Undefined) {
693         return false;
694     } else if (r.type == Undefined) {
695         return true;
696     }
697     return false;
698 }
699 bool operator>=(const AnalysisNode& l, const AnalysisNode& r) {
700     return (l > r || l == r);
701 }
702 bool operator<(const AnalysisNode& l, const AnalysisNode& r) {
703     return (r > l);
704 }
705 bool operator<=(const AnalysisNode& l, const AnalysisNode& r) {
706     return (l < r || l == r);
707 }
708 
709 string AnalysisNode::ToString() {
710     string str = "";
711     if (type == Logarithmic) {
712         str = "log(n)";
713         if (exponent != 1) {
714             str = "(" + str + ")^" + to_string(exponent);
715         }
716     } else if (type == Exponential) {
717         str = "n";
718         if (exponent != 1) {
719             str += "^" + to_string(exponent);
720         }
721     } else if (type == Constant) {
722         str = "C";
723     }
724     return str;
725 }
726 
727 void AnalysisNode::SetToUndefined() {
728     this->base = 1;
729     this->exponent = 1;
730     this->type = Undefined;
731 }
732 void AnalysisNode::SetToConstant() {
733     this->base = 1;
734     this->exponent = 1;
735     this->type = Constant;
736 }
```

```
737 void AnalysisNode::SetToExponential(int exponent){
738     this->base = 1;
739     this->exponent = exponent;
740     this->type = Exponential;
741 }
742 void AnalysisNode::SetToLogarithmic(int base, int exponent){
743     this->base = base;
744     this->exponent = exponent;
745     this->type = Logarithmic;
746 }
```

```
 1 /*
 2  *  AnalyzeModule.h
 3  *
 4  *
 5  *  Created: 3/24/2017 by Ryan Tedeschi
 6  */
 7
 8 #ifndef ANALYZEMODULE_H
 9 #define ANALYZEMODULE_H
10
11 #include <string>
12 #include <vector>
13 #include <iostream>
14 #include "../../shared/CASP_Plugin/CASP_Plugin.h"
15
16 using namespace std;
17
18 enum NodeType { Constant, Exponential, Logarithmic, Undefined };
19
20 class AnalysisNode {
21     public:
22         AnalysisNode();
23
24         void SetToUndefined();
25         void SetToConstant();
26         void SetToExponential(int exponent);
27         void SetToLogarithmic(int base, int exponent);
28
29         string ToString();
30
31         int exponent = 1;
32         int base = 1;
33         NodeType type = Undefined;
34
35         friend bool operator==(const AnalysisNode&, const AnalysisNode&);
36         friend bool operator!=(const AnalysisNode&, const AnalysisNode&);
37         friend bool operator>(const AnalysisNode&, const AnalysisNode&);
38         friend bool operator>=(const AnalysisNode&, const AnalysisNode&);
39         friend bool operator<(const AnalysisNode&, const AnalysisNode&);
40         friend bool operator<=(const AnalysisNode&, const AnalysisNode&);
41         AnalysisNode& operator*(AnalysisNode&);
42         AnalysisNode& operator=(AnalysisNode&);
43         AnalysisNode* operator=(AnalysisNode*);
44
45     private:
46
47 };
48
49 class Analysis {
50
51     public:
52         Analysis();
53         void AddFactor(AnalysisNode*);
54         void AddConstantFactor();
55         void AddExponentialFactor(int);
56         void AddLogarithmicFactor(int, int);
57         string ToString();
58
59         friend bool operator==(const Analysis&, const Analysis&);
60         friend bool operator!=(const Analysis&, const Analysis&);
61         friend bool operator>(const Analysis&, const Analysis&);
62         friend bool operator>=(const Analysis&, const Analysis&);
63         friend bool operator<(const Analysis&, const Analysis&);
64         friend bool operator<=(const Analysis&, const Analysis&);
65         Analysis& operator*(Analysis&);
66
```

```
67          bool IsUndefined();
68
69      private:
70          AnalysisNode* constant = NULL;
71          AnalysisNode* exponential = NULL;
72          AnalysisNode* logarithmic = NULL;
73
74          bool undefined = false;
75  };
76
77  class AnalysisTree {
78      public:
79          AnalysisTree();
80
81          void AddChild(AnalysisTree*);
82          void AddFactor(AnalysisNode*);
83          void AddConstantFactor();
84          void AddExponentialFactor(int);
85          void AddLogarithmicFactor(int, int);
86          void SetAnalysis(Analysis*);
87
88          Analysis* GetAnalysis();
89
90      private:
91
92          vector<AnalysisTree*> children;
93          Analysis* analysis = NULL;
94
95  };
96
97  class AnalyzeModule : public CASP_Plugin {
98      public:
99          AnalyzeModule();
100
101         virtual CASP_Return* Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn = NULL);
102
103     private:
104         void GetAllAnalyses(Markup*);
105         Analysis* GetRootAnalysis(vector<Markup*>);
106         Analysis* GetFunctionAnalysis(Markup*);
107
108         void analyzeProcess(Markup*, AnalysisTree*);
109         void analyzeMethodCall(Markup*, AnalysisTree*);
110         void analyzeDecision(Markup*, AnalysisTree*);
111         void analyzeLoop(Markup*, AnalysisTree*);
112         void processStatement(Markup*, AnalysisTree*);
113         void processBlock(Markup*, AnalysisTree*);
114
115         unordered_map<string, Analysis*> functionTable;
116         unordered_map<string, Markup*> markupTable;
117
118 };
119
120 #endif
```

```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 }
```

```
/*
 *  OutlineModule.h
 *
 *
 *  Created: 3/24/2017 by Ryan Tedeschi
 */

#ifndef OUTLINEMODULE_H
#define OUTLINEMODULE_H

#include <string>
#include <iostream>
#include <vector>
#include "../../shared/CASP_Plugin/CASP_Plugin.h"
#include "../../shared/Printable/Printable.h"

using namespace std;

enum EntryType { Start, MethodCall, Process, Loop, Decision, EndDecision, IO, End };
class OutlineModule;
class Outline;
class Node;
class Edge;

class OutlineModule : public CASP_Plugin {
    public:
        OutlineModule();
        virtual CASP_Return* Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn = NULL);

    private:
        vector<Outline*> GetAllOutlines(Markup*);
        Outline* GetRootOutline(vector<Markup*>);
        Outline* GetFunctionOutline(Markup*);
        CASP_Return* FormatData(vector<Outline*>);

        Node* stripProcess(Markup*, Outline*, Node*, string = "");
        Node* stripMethodCall(Markup*, Outline*, Node*, string = "");
        Node* stripDecision(Markup*, Outline*, Node*, string = "");
        Node* stripFor(Markup*, Outline*, Node*, string = "");
        Node* stripWhile(Markup*, Outline*, Node*, string = "");
        Node* processStatement(Markup*, Outline*, Node*, string = "");
        Node* processBlock(Markup*, Outline*, Node*, string = "");
};

class Outline : public Printable {
    public:
        Outline();

        Node* AppendBlock(EntryType, string, Node*);
        Node* AppendBlock(EntryType, string, Node*, string);
        Node* AppendBlock(Node*);
        void Print();

        GenericArray* Output();

    private:
        vector<Node*> nodes;
        int maxId = 0;
        Node* head = NULL;
};

class Node : public Printable {
    public:
        Node(string, EntryType, int);

        Edge* AddEdgeTo(Node*);
```

```
67          Edge* AddEdgeTo(Node*, string);
68          Edge* AddEdgeFrom(Node*);
69          Edge* AddEdgeFrom(Node*, string);
70          void Print();
71
72          GenericObject* Output();
73
74          int id;
75          string data;
76          EntryType type;
77
78      private:
79          vector<Edge*> edges;
80
81  };
82
83  class Edge : public Printable {
84      public:
85          Edge(Node*, Node*, string);
86          Edge(Node*, Node*);
87          void Print();
88
89          GenericObject* Output();
90
91          string data = "";
92          Node* source = NULL;
93          Node* target = NULL;
94
95      private:
96  };
97
98  #endif
```

```cpp
#include "PrintModule.h"

static string _PrintModule = RegisterPlugin("Print", new PrintModule());

PrintModule::PrintModule() {}

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

    markup->Print();

    Tree* outputTree = GenerateTree(markup);
    returnData->Data()->Add("ParseTree", outputTree->Output());

    return returnData;

}

Tree* PrintModule::GenerateTree(Markup* m) {
    vector<Markup*> children = m->Children();
    Tree* t = new Tree();
    t->Title = m->GetID();

    if (children.size() > 0) {
        for (int i = 0; i < children.size(); i++) {
            t->Children.push_back(GenerateTree(children[i]));
        }
    } else {
        t->Data = m->GetData();
    }

    return t;
}


GenericObject* Tree::Output() {

    GenericArray* children = new GenericArray();

    for (int i = 0; i < Children.size(); i++) {
        children->Add(Children[i]->Output());
    }

    return new GenericObject({
        {"Data", CreateLeaf(Data)},
        {"Title", CreateLeaf(Title)},
        {"Children", children}
    });

}
```

```
1 /*
2  *  PrintModule.h
3  *
4  *
5  *  Created: 4/2/2017 by Ryan Tedeschi
6  */
7
8 #ifndef PRINTMODULE_H
9 #define PRINTTMODULE_H
10
11 #include <string>
12 #include <iostream>
13 #include "../../shared/CASP_Plugin/CASP_Plugin.h"
14
15 using namespace std;
16
17 class Tree {
18     public:
19         string Title;
20         string Data;
21
22         vector<Tree*> Children;
23
24         GenericObject* Output();
25 };
26
27 class PrintModule : public CASP_Plugin {
28     public:
29         PrintModule();
30
31         virtual CASP_Return* Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn = NULL);
32
33     private:
34         Tree* GenerateTree(Markup*);
35 };
36
37
38 #endif
```

```cpp
#include "TranslateModule.h"

static string _TranslateModule = RegisterPlugin("Translate", new TranslateModule());

TranslateModule::TranslateModule() {}

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

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

    bool languageRead = true;
    this->source_ldo = source_ldo;
    string targetLanguage = Helpers::ParseArgument("targetlang", fnArgs);

    if (targetLanguage != "") {

        try {
            ReadLanguageFile(targetLanguage);
        } catch (...) {
            returnData->AddStandardError("Language '" + targetLanguage + "' could not be read. Could not proceed with translation.");
            languageRead = false;
        }

        if (languageRead) {
            try {
                Translate(markup);
            } catch (...) {
                returnData->AddStandardError("Error while processing translation.");
            }
        }

    } else {
        returnData->AddStandardError("Target language not provided. Make sure to use argument 'targetlang'.");
    }

    return returnData;

}

void TranslateModule::Translate(Markup* markup) {

    Markup* targetRoot = new Markup("ROOT");
    string nodeId = markup->GetID();

    vector<Markup*> children = markup->Children();
    for (int i = 0; i < children.size(); i++) {
        Markup* c = MatchTargetProd(children[i]);
        if (c != NULL)
            targetRoot->AddChild(c);
    }

    // markup->Print();
    // targetRoot->Print();

    vector<Token> tl1 = source_ldo->Tokenize(markup);
    vector<Token> tl2 = target_ldo->Tokenize(targetRoot);

    cout << endl << PrettyPrint(tl1) << endl << endl;
    cout << endl << PrettyPrint(tl2) << endl << endl;

    returnData->Data()->Add("OriginalSource", CreateObject({
        { "Language" , CreateLeaf(source_ldo->GetLanguage()) },
        { "Data" , CreateLeaf(PrettyPrint(tl1)) }
    }));

```

```
67      returnData->Data()->Add("TranslatedSource", CreateObject({
68          { "Language" , CreateLeaf(target_ldo->GetLanguage()) },
69          { "Data" , CreateLeaf(PrettyPrint(tl2)) }
70      }));
71
72  }
73
74  string TranslateModule::PrettyPrint(vector<Token> tokens) {
75      int i = 0;
76      return PrintBlockBody(tokens, &i, 0);
77  }
78
79  string TranslateModule::PrintBlockBody(vector<Token> tokens, int* index, int tabIndex) {
80      string str = "";
81      int i;
82      int size = tokens.size();
83      bool finishedBlock = false;
84      bool endStmt = false;
85      bool forStmts = false;
86
87      for (i = *index; i < size; i++) {
88          Token t = tokens[i];
89
90          if (t.id == "L_CU_BRACKET") {
91              if (i != 0)
92                  str += "\n";
93              str += Helpers::DupStr("    ", tabIndex);
94              str += t.value;
95              str += "\n" + Helpers::DupStr("    ", tabIndex + 1);
96              i++;
97              str += PrintBlockBody(tokens, &i, tabIndex + 1);
98              i--;
99              finishedBlock = true;
100             endStmt = false;
101             forStmts = false;
102             continue;
103         } else if (t.id == "R_CU_BRACKET") {
104             if (!finishedBlock)
105                 break;
106             str += "\n";
107             str += Helpers::DupStr("    ", tabIndex);
108             str += t.value;
109             str += "\n";
110             str += Helpers::DupStr("    ", tabIndex);
111             endStmt = false;
112         } else if (t.id == "SEMICOLON" && !forStmts) {
113             str += t.value;
114             endStmt = true;
115         } else {
116             if (endStmt) {
117                 str += "\n" + Helpers::DupStr("    ", tabIndex);
118                 endStmt = false;
119             }
120             if (t.id == "FOR")
121                 forStmts = true;
122             str += t.value + " ";
123         }
124         finishedBlock = false;
125     }
126     *index = i;
127     return str;
128 }
129
130
131 Markup* TranslateModule::MatchTargetProd(Markup* markup) {
132
133     string nodeId = markup->GetID();
```

```
134
135     if (!markup->IsLeaf()) {
136         Production* targetProd = target_ldo->findProdById(nodeId);
137
138         if (targetProd != NULL) {
139             return TranslateProd(markup, targetProd);
140         } else {
141             Markup* ret = new Markup(nodeId);
142             Markup* t = NULL;
143             returnData->AddStandardWarning("No matching translation for construct '" + nodeId + "'");
144             // add warning that this node could not be translated
145             // vector<Markup*> children = markup->Children();
146             // for (int i = 0; i < children.size(); i++) {
147             //     t = MatchTargetProd(children[i]);
148             //     if (t != NULL) {
149             //         ret->AddChild(t);
150             //     }
151             // }
152             // return ret;
153             return NULL;
154         }
155     } else {
156         string nodeValue = markup->GetData();
157         bool dynamicTerminal = source_ldo->LookupTerminalValue(nodeId) == "";
158         if (!dynamicTerminal) {
159             string newTerminal = target_ldo->LookupTerminalValue(nodeId);
160             if (newTerminal != "") {
161                 return new Markup(nodeId, newTerminal);
162             } else {
163                 returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
164                 // add warning that there is no translation
165                 return NULL;//new Markup(nodeId, nodeValue);
166             }
167         } else {
168             returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
169             // add warning that this cannot be translated
170             return NULL;// new Markup(nodeId, nodeValue);
171         }
172
173     }
174
175     return NULL;
176 }
177
178 Markup* TranslateModule::TranslateProd(Markup* source, Production* target) {
179     Markup* newMarkup = new Markup(target->GetId());
180     vector<ProductionSet*> children = target->GetRootProductionSet()->GetChildren();
181
182     for (int i = 0; i < children.size(); i++) {
183         ProductionSet* p = children[i];
184         Markup* c = NULL;
185
186         switch (p->GetType()) {
187             case _Terminal:
188                 c = HandleTerminal(source, p, true);
189                 break;
190             case _Production:
191                 c = HandleProduction(source, p, true);
192                 break;
193             case _Alternation:
194                 c = HandleAlternation(source, p);
195                 break;
196         }
197
198         if (c != NULL) {
199             newMarkup->AddChild(c);
200         }
```

```
201     }
202
203     return newMarkup;
204 }
205 Markup* TranslateModule::HandleTerminal(Markup* source, ProductionSet* set, bool fillInOnNoMatch) {
206     string nodeId = set->GetSource();
207     Markup* sourceTerminal = source->FindFirstChildById(nodeId);
208     string newTerminal = target_ldo->LookupTerminalValue(nodeId);
209
210     if (fillInOnNoMatch || sourceTerminal != NULL) {
211         if (newTerminal != "") {
212             return new Markup(nodeId, newTerminal);
213         } else if (sourceTerminal != NULL) {
214             // returnData->AddStandardWarning("The translation for terminal '" + nodeId + "' (value = '" + sourceTerminal->GetData() + "') is not guaranteed. Check syntax.");
215             // add warning that this is an inconclusive translation
216             return new Markup(nodeId, sourceTerminal->GetData());
217         }
218     }
219     // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
220     // add warning that there is no matching translation
221     return NULL;
222 }
223 Markup* TranslateModule::HandleProduction(Markup* source, ProductionSet* set, bool dummyOnFail) {
224     string nodeId = set->GetSource();
225     Markup* ret = NULL;
226
227     Markup* sourceProduction = source->FindFirstChildById(nodeId);
228     if (sourceProduction != NULL) {
229         Production* targetProd = target_ldo->findProdById(nodeId);
230         ret = TranslateProd(sourceProduction, targetProd);
231         if (ret != NULL) {
232             return ret;
233         }
234     } else {
235
236     }
237     if (dummyOnFail && set->GetMultiplicity() != "?")
238         ret = new Markup(nodeId, "<" + nodeId + ">");
239     // returnData->AddStandardWarning("No matching translation for production '" + nodeId + "'");
240     // add warning that there is no matching translation
241     return ret;
242 }
243 Markup* TranslateModule::HandleAlternation(Markup* source, ProductionSet* set) {
244     Markup* newMarkup = NULL;
245     vector<ProductionSet*> children = set->GetChildren();
246
247     for (int i = 0; i < children.size() && newMarkup == NULL; i++) {
248         ProductionSet* p = children[i];
249
250         switch (p->GetType()) {
251             case _Terminal:
252                 newMarkup = HandleTerminal(source, p, false);
253                 break;
254             case _Production:
255                 newMarkup = HandleProduction(source, p, false);
256                 break;
257             case _Alternation:
258                 newMarkup = HandleAlternation(source, p);
259                 break;
260         }
261     }
262
263     if (newMarkup == NULL) {
264         // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
265     }
266
267     return newMarkup;
```

```
268 }
269
270
271 void TranslateModule::ReadLanguageFile(string targetLanguage) {
272     // read and parse file;
273     target_ldo = new LanguageDescriptorObject(targetLanguage);
274 }
```

```
/*
 *  TranslateModule.h
 *
 *
 *  Created: 3/24/2017 by Ryan Tedeschi
 */

#ifndef TRANSLATEMODULE_H
#define TRANSLATEMODULE_H

#include <string>
#include <iostream>
#include "../../shared/CASP_Plugin/CASP_Plugin.h"

using namespace std;

class TranslateModule : public CASP_Plugin {
    public:
        TranslateModule();

        virtual CASP_Return* Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn = NULL);

    private:

        string PrettyPrint(vector<Token>);
        string PrintBlockBody(vector<Token>, int*, int);

        void ReadLanguageFile(string);
        void Translate(Markup*);
        Markup* MatchTargetProd(Markup*);
        Markup* TranslateProd(Markup*, Production*);

        Markup* HandleTerminal(Markup*, ProductionSet*, bool);
        Markup* HandleProduction(Markup*, ProductionSet*, bool);
        Markup* HandleAlternation(Markup*, ProductionSet*);

        LanguageDescriptorObject* target_ldo = NULL;
        LanguageDescriptorObject* source_ldo = NULL;

};

#endif
```

```
 1 /*
 2  *  plugins.h
 3  *  Enumerates all active plugin source code
 4  *
 5  *  Created: 3/24/2017 by Ryan Tedeschi
 6  */
 7
 8 #ifndef PLUGINS_H
 9 #define PLUGINS_H
10
11 #include "OutlineModule/OutlineModule.h"
12
13 #endif
```

```
#include "CASP_Plugin.h"

unordered_map<string, CASP_Plugin*> plugins;

string RegisterPlugin(string id, CASP_Plugin* plugin) {
    std::transform(id.begin(), id.end(), id.begin(), ::tolower);
    plugins[id] = plugin;

    return id;
}

CASP_Plugin* GetModule(string id) {
    id = Helpers::toLower(id);
    CASP_Plugin* plugin = NULL;
    if (ModuleExists(id)) {
        plugin = plugins[id];
    } else {
        throw "Module '" + id + "' does not exist.";
    }
    return plugin;
}

bool ModuleExists(string id) {
    std::transform(id.begin(), id.end(), id.begin(), ::tolower);
    CASP_Plugin* p = plugins[id];
    if (p == NULL) {
        plugins.erase(id);
        return false;
    }
    return true;
}
```

```
/*
 *  CASP_Plugin.h
 *  Defines the base class for a Plugin Module
 *
 *  Created: 3/24/2017 by Ryan Tedeschi
 */

#ifndef CASP_PLUGIN_H
#define CASP_PLUGIN_H

#include <algorithm>
#include <string>
#include <vector>
#include <unordered_map>
#include "../Markup/Markup.h"
#include "../CASP_Return/CASP_Return.h"
#include "../LanguageDescriptor/LanguageDescriptor.h"

using namespace std;

class CASP_Plugin {
    public:
        virtual CASP_Return* Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn = NULL) = 0;

        CASP_Return* returnData = NULL;
};

// extern unordered_map<string, CASP_Plugin*> plugins;
string RegisterPlugin(string, CASP_Plugin*);
CASP_Plugin* GetModule(string);
bool ModuleExists(string);

#endif
```

```cpp
#include "CASP_Return.h"

void GenericData::Print() {}

GenericObject::GenericObject() {}
GenericObject::GenericObject(unordered_map<string, GenericData*> map) {
    data = map;
}
void GenericObject::Print() {
    int count = 0;
    cout << "{";
    for (auto it = data.begin(); it != data.end(); ++it ) {
        if (count++ > 0) {
            cout << ",";
        }
        cout << "\"" << it->first << "\":";
        if (it->second != NULL)
            it->second->Print();
        else
            cout << "null";
    }
    cout << "}";
}

void GenericObject::Add(string key, GenericData* d) {
    data[key] = d;
}

GenericData* GenericObject::At(string key) {
    return data[key];
}


GenericArray::GenericArray() {}
GenericArray::GenericArray(vector<GenericData*> list) {
    data = list;
}

void GenericArray::Print() {
    cout << "[";
    for (int i = 0; i < data.size(); i++) {
        if (i > 0)
            cout << ",";
        data[i]->Print();
    }
    cout << "]";
}

void GenericArray::Add(GenericData* d) {
    data.push_back(d);
}

GenericData* GenericArray::At(int index) {
    return data[index];
}

CASP_Return::CASP_Return() {
    Add("Data", new GenericObject());
    Add("Warnings", new GenericArray());
    Add("Errors", new GenericArray());
}

GenericArray* CASP_Return::Errors() {
    return (GenericArray*)data["Errors"];
}
GenericArray* CASP_Return::Warnings() {
```

```
67     return (GenericArray*)data["Warnings"];
68 }
69 GenericObject* CASP_Return::Data() {
70     return (GenericObject*)data["Data"];
71 }
72 void CASP_Return::AddStandardWarning(string message, int warningId) {
73     GenericObject* warn = CreateObject({ { "id", CreateLeaf(warningId) }, { "message", CreateLeaf(message) } });
74     Warnings()->Add(warn);
75 }
76 void CASP_Return::AddStandardError(string message, int errorId) {
77     GenericObject* err = CreateObject({ { "id", CreateLeaf(errorId) }, { "message", CreateLeaf(message) } });
78     Errors()->Add(err);
79 }
80
81
82 GenericObject* CreateObject() {
83     GenericObject* ob = new GenericObject();
84     return ob;
85 };
86 GenericObject* CreateObject(unordered_map<string, GenericData*> map) {
87     GenericObject* ob = new GenericObject(map);
88     return ob;
89 };
90 GenericArray* CreateArray() {
91     GenericArray* arr = new GenericArray();
92     return arr;
93 };
94 GenericArray* CreateArray(vector<GenericData*> list) {
95     GenericArray* arr = new GenericArray(list);
96     return arr;
97 };
```

```
 1 /*
 2  *  CASP_Return.h
 3  *  Defines a generic return object for the program
 4  *
 5  *  Created: 4/4/2017 by Ryan Tedeschi
 6  */
 7
 8 #ifndef CASP_RETURN_H
 9 #define CASP_RETURN_H
10
11 #include <iostream>
12 #include <algorithm>
13 #include <string>
14 #include <unordered_map>
15 #include <vector>
16 #include "../Printable/Printable.h"
17
18 using namespace std;
19
20 class GenericData : public Printable {
21     public:
22         string GetType() {
23             return type;
24         };
25         virtual void Print();
26
27     private:
28         string type = "";
29 };
30
31 template<typename T>
32 class GenericLeaf : public GenericData {
33     public:
34         GenericLeaf(T data) {
35             this->data = data;
36         };
37
38         virtual void Print() {
39             try {
40                 cout << specialLookups.at(data);
41             } catch (...) {
42                 cout << data;
43             }
44         };
45
46         void Assign(T data) {
47             this->data = data;
48         };
49
50         void AddSpecial(T input, string output) {
51             specialLookups[input] = output;
52         };
53
54     protected:
55         string type = "Leaf";
56         T data;
57         unordered_map<T, string> specialLookups;
58 };
59
60 template<typename T>
61 static inline
62 GenericLeaf<T>* CreateLeaf(T data) {
63     GenericLeaf<T>* leaf = new GenericLeaf<T>(data);
64     return leaf;
65 };
66 template<>
```

```
67 static inline
68 GenericLeaf<bool>* CreateLeaf(bool data) {
69     GenericLeaf<bool>* leaf = new GenericLeaf<bool>(data);
70     leaf->AddSpecial(true, "true");
71     leaf->AddSpecial(false, "false");
72     return leaf;
73 };
74 template<>
75 static inline
76 GenericLeaf<string>* CreateLeaf<string>(string data) {
77 
78     int index = -1;
79     while ((index = data.find("\"", index + 1)) != -1) {
80         data = data.substr(0, index) + "\\" + data.substr(index, data.size());
81 
82         index++;
83     }
84 
85     GenericLeaf<string>* leaf = new GenericLeaf<string>("\"" + data + "\"");
86     return leaf;
87 };
88 
89 class GenericObject : public GenericData {
90     public:
91         GenericObject();
92         GenericObject(unordered_map<string, GenericData*>);
93         virtual void Print();
94         void Add(string, GenericData*);
95         GenericData* At(string);
96 
97     protected:
98         string type = "Object";
99         unordered_map<string, GenericData*> data;
100 
101 };
102 GenericObject* CreateObject();
103 GenericObject* CreateObject(unordered_map<string, GenericData*>);
104 
105 class GenericArray : public GenericData {
106     public:
107         GenericArray();
108         GenericArray(vector<GenericData*>);
109         virtual void Print();
110         void Add(GenericData*);
111         GenericData* At(int);
112 
113     protected:
114         string type = "Array";
115         vector<GenericData*> data;
116 };
117 
118 GenericArray* CreateArray();
119 GenericArray* CreateArray(vector<GenericData*>);
120 
121 class CASP_Return : public GenericObject {
122     public:
123         CASP_Return();
124 
125         GenericArray* Errors();
126         GenericArray* Warnings();
127         GenericObject* Data();
128 
129         void AddStandardWarning(string, int = -1);
130         void AddStandardError(string, int = -1);
131 
132     private:
133 };
```

```
134
135
136 #endif
```

```cpp
/*
 *  Helpers.h
 *  Defines Helper functions for the application
 *
 *
 *  Created: 2/7/2017 by Ryan Tedeschi
 */

#include "Helpers.h"

using namespace std;

namespace Helpers {
    string ReadFile(string filename) {
        FILE* fp = fopen(filename.c_str(), "r");
        string filetext = "";

        if (fp != NULL) {
            char c;
            while ((c = fgetc(fp)) != EOF) {
                filetext += c;
            }
        }

        return filetext;
    }

    string DupStr(string str, int count) {
        string s= "";
        for (int i = 0; i < count; i++) {
            s += str;
        }
        return s;
    }

    string toLower(string str) {
        string c(str);
        std::transform(c.begin(), c.end(), c.begin(), ::tolower);
        return c;
    }
    string toUpper(string str) {
        string c(str);
        std::transform(c.begin(), c.end(), c.begin(), ::toupper);
        return c;
    }

    vector<string> ParseArrayArgument(string tag, vector<arg> args) {
        tag = toLower(tag);
        vector<string> ls;

        for (int i = 0; i < args.size(); i++) {
            if (args[i].id == tag)
                ls.push_back(args[i].value);
        }

        return ls;
    }

    string ParseArgument(string tag, vector<arg> args) {
        tag = toLower(tag);
        string s = "";

        for (int i = 0; i < args.size(); i++) {
            if (args[i].id == tag) {
                s = args[i].value;
                break;
```

```
67             }
68         }
69 
70         return s;
71     }
72 }
```

```
/*
 *  LanguageDescriptor.h
 *  Defines Helper functions for the application
 *
 *
 *  Created: 2/7/2017 by Ryan Tedeschi
 */

#ifndef HELPERS_H
#define HELPERS_H

#include <string>
#include <vector>
#include <unordered_map>
#include <list>
#include <algorithm>

using namespace std;

struct arg {
    arg(string id, string value) {
        this->id = id;
        this->value = value;
    };
    string id;
    string value;
};

namespace Helpers {
    string ReadFile(string);
    string DupStr(string, int);

    vector<string> ParseArrayArgument(string, vector<arg>);
    string ParseArgument(string, vector<arg>);

    string toLower(string);
    string toUpper(string);

    template<typename T>
    vector<T> concat(vector<T> source, vector<T> addition) {
        source.insert(source.end(), addition.begin(), addition.end());
        return source;
    };
};

template<class T>
class State {
    public:
        State(string id) {
            this->id = id;
        };
        void SetGoal(string token) {
            this->acceptingToken = token;
            isFinal = true;
        };
        void UnsetGoal() {
            this->acceptingToken;
            isFinal = false;
        };
        void AddTransition(State<T>* target, vector<T> input) {
            this->transitionInputs.push_back(input);
            this->transitionStates.push_back(target);
            for (int i = 0; i < input.size(); i++) {
                transitions[input[i]] = target;
            }
        };
```

```
67          State<T>* Transition(T input) {
68              return transitions[input];
69          };
70          string GetId() {
71              return id;
72          };
73          string GetToken() {
74              return acceptingToken;
75          };
76          bool IsGoal() {
77              return isFinal;
78          };
79          void Print() {
80              cout << "State '" << id << "'";
81              if (isFinal)
82                  cout << " (GOAL - '" << acceptingToken << "')";
83              for (int i = 0; i < transitionStates.size(); i++) {
84                  cout << "\n\tTransitions to state '" << transitionStates[i]->GetId() << "' with inputs ";
85                  for (int j = 0; j < transitionInputs[i].size(); j++) {
86                      if (j > 0)
87                          cout << ", ";
88                      cout << transitionInputs[i][j];
89                  }
90              }
91              cout << endl;
92          };
93
94      private:
95          unordered_map<T, State<T>*> transitions;
96          vector<State<T>*> transitionStates;
97          vector<vector<T>> transitionInputs;
98          string id = "";
99          string acceptingToken = "";
100         bool isFinal = false;
101 };
102
103 template<class T>
104 class FSM {
105     public:
106         FSM() {};
107
108         State<T>* AddState(string id) {
109             if (!HasState(id)) {
110                 State<T>* newState = new State<T>(id);
111                 states[id] = newState;
112                 return newState;
113             }
114             return NULL;
115         };
116         void AddTransition(string start, string target, vector<T> transitionInput) {
117             State<T>* Start = GetState(start);
118             State<T>* Target = GetState(target);
119             if (Start != NULL && Target != NULL) {
120                 Start->AddTransition(Target, transitionInput);
121             }
122         };
123         void SetInitialState(string id) {
124             initialState = GetState(id);
125         };
126         void AddGoal(string id, string token) {
127             State<T>* state = GetState(id);
128             if (state != NULL) {
129                 state->SetGoal(token);
130             }
131         };
132         void RemoveGoal(string id) {
133             State<T>* state = GetState(id);
```

```
134             if (state != NULL) {
135                 state->UnsetGoal();
136             }
137         };
138         bool HasState(string id) {
139             return GetState(id) != NULL;
140         };
141         State<T>* GetState(string id) {
142             return states[id];
143         };
144         string Transition(T input) {
145             string ret = "";
146             if (currentState != NULL) {
147                 State<T>* nextState = currentState->Transition(input);
148                 if (nextState != NULL) {
149                     currentState = nextState;
150                 } else {
151                     if (currentState->IsGoal()) {
152                         ret = currentState->GetToken();
153                         Reset();
154                     } else
155                         ret = "ERROR";
156                 }
157             } else {
158                 ret = "ERROR";
159                 Reset();
160             }
161             return ret;
162         };
163         State<T>* CurrentState() {
164             return currentState;
165         };
166         void Reset() {
167             currentState = initialState;
168         };
169 
170         void Print() {
171             cout << "---- FINITE STATE MACHINE ----\n";
172             cout << "Initial State: " << initialState->GetId() << endl;
173 
174             for ( auto it = states.begin(); it != states.end(); ++it )
175                 it->second->Print();
176         };
177 
178     private:
179         unordered_map<string, State<T>*> states;
180         State<T>* initialState = NULL;
181         State<T>* currentState = NULL;
182 };
183 
184 #endif
```

```cpp
/*
 *  LanguageDescriptor.h
 *  Defines the Language Descriptor class, which is the bridge between a text language descriptor file
 *
 *
 *  Created: 1/3/2017 by Ryan Tedeschi
 */

#include "LanguageDescriptor.h"

using namespace std;

Token::Token(string id, string value) {
    this->id = id;
    this->value = value;
};

void Token::Print() {
    cout << "[" << id << "]\t" << value << endl;
};

string LanguageDescriptorObject::LookupTerminalValue(string terminalID) {
    return terminals[terminalID];
};

bool LanguageDescriptorObject::IsTerminalIgnored(string terminalID) {
    try {
        return ignore.at(terminalID);
    } catch (...) {
        return false;
    }
}


void LanguageDescriptorObject::ParseTerminalValues(string data) {

    string t = string(data);
    regex r = regex("T\\([ \t]*(.+)[ \t]*,[ \t]*\"(.*)\"[ \t]*\\)");
    smatch matches;

    while (regex_search(t, matches, r)) {
        string terminalID = matches[1].str();
        string terminalValue = matches[2].str();
        terminals[terminalID] = terminalValue;

        t = matches.suffix().str();
    }
}

void LanguageDescriptorObject::ParseReservedWords(string data) {

    string t = string(data);
    regex r = regex("ReservedWord\\([ \t]*(.+)[ \t]*,[ \t]*(.+)[ \t]*\\)");
    smatch matches;

    while (regex_search(t, matches, r)) {
        string terminalValue = matches[1].str();
        string terminalID = matches[2].str();
        reservedWords[terminalValue] = terminalID;
        terminals[terminalID] = terminalValue;

        t = matches.suffix().str();
    }
}

void LanguageDescriptorObject::ParseIgnores(string data) {
```

```
 67
 68     string t = string(data);
 69     regex r = regex("Ignore\\([ \t]*(.+)[ \t]*\\)");
 70     smatch matches;
 71
 72     while (regex_search(t, matches, r)) {
 73         string terminalID = matches[1].str();
 74         ignore[terminalID] = true;
 75
 76         t = matches.suffix().str();
 77     }
 78 }
 79
 80 void LanguageDescriptorObject::ParseFSM(string data) {
 81
 82     string t = string(data);
 83     regex r = regex("^\\([ \t]*([a-zA-Z_0-9]+)[ \t]*,[ \t]*([^\t\n]+)[ \t]*\\)[ \t]*->[ \t]*([^ \t\n]+)$");
 84     smatch matches;
 85
 86     while (regex_search(t, matches, r)) {
 87         string fromState = matches[1].str();
 88         string toState = matches[3].str();
 89         string chars = matches[2].str();
 90
 91         int index = -1;
 92         while ((index = chars.find("\\", index + 1)) != -1) {
 93             if (index < chars.size() - 1) {
 94                 chars = chars.substr(0, index) + chars.substr(index + 1, chars.size());
 95                 switch (chars[index]) {
 96                     case 'n':
 97                         chars[index] = '\n';
 98                         break;
 99                     case 't':
100                         chars[index] = '\t';
101                         break;
102                     case 'r':
103                         chars[index] = '\r';
104                         break;
105                     case '0':
106                         chars[index] = '\0';
107                         break;
108                 }
109             } else
110                 chars = chars.substr(0, index);
111         }
112
113         vector<char> stateTransitions;
114         for (int i = 0; i < chars.size(); i++) {
115             stateTransitions.push_back(chars[i]);
116         }
117
118         stateMachine.AddState(fromState);
119         stateMachine.AddState(toState);
120         stateMachine.AddTransition(fromState, toState, stateTransitions);
121
122         // cout << "State " << matches[1] << " moves to state " << matches[3] << " with any of the following input: " << matches[2] << endl;
123         t = matches.suffix().str();
124     }
125
126     t = string(data);
127     r = regex("^F\\([ \t]*([^ \t\n]+)[ \t]*,[ \t]*([^ \t\n]+)[ \t]*\\)[ \t]*$");
128
129     while (regex_search(t, matches, r)) {
130         string target = matches[1].str();
131         string token = matches[2].str();
132
133         stateMachine.AddGoal(target, token);
```

```
134
135         // cout << "State " << matches[1] << " accepts token " << matches[2] << endl;
136         t = matches.suffix().str();
137     }
138
139     t = data;
140     r = regex("^I\\([ \t]*([^ \t\n]+)[ \t]*\\)$");
141
142     if (regex_search(t, matches, r)) {
143         string target = matches[1].str();
144
145         stateMachine.SetInitialState(target);
146
147         // cout << "State " << matches[1] << " is the initial state" << endl;
148         t = matches.suffix().str();
149     }
150
151     // stateMachine.Print();
152
153 }
154
155 vector<Token> LanguageDescriptorObject::Tokenize(string input) {
156     vector<Token> tokens;
157     string token;
158     string tokenData;
159
160     stateMachine.Reset();
161
162     for (int i = 0; i < input.size(); i++) {
163         tokenData += input[i];
164         if ((token = stateMachine.Transition(input[i])) != "") {
165             if (token == "ERROR") {
166                 if (input[i] != ' ' && input[i] != '\n' && input[i] != '\r' && input[i] != '\t')
167                     cout << "State machine encountered an error on character '" << input[i] << "'\n";
168             } else {
169                 tokenData.pop_back();
170
171                 if (reservedWords[tokenData] != "")
172                     token = reservedWords[tokenData];
173
174                 if (!IsTerminalIgnored(token))
175                     tokens.push_back(Token(token, tokenData));
176                 else
177                     cout << "Ignoring terminal " << token << ", value = \"" << tokenData << "\"" << endl;
178                 i--;
179             }
180             tokenData = "";
181         }
182     }
183
184     if (token == "") {
185         // accept the last token, only if there is one to accept
186         token = stateMachine.Transition('\0');
187         if (token == "" || token == "ERROR") {
188             cout << "State machine encountered an error on character 'EOF'\n";
189         } else {
190             if (reservedWords[tokenData] != "")
191                 token = reservedWords[tokenData];
192
193             if (!IsTerminalIgnored(token))
194                 tokens.push_back(Token(token, tokenData));
195             else
196                 cout << "Ignoring terminal " << token << ", value = \"" << tokenData << "\"" << endl;
197         }
198     }
199     stateMachine.Reset();
200
```

```
201     // for (int i = 0; i < tokens.size(); i++) {
202     //     tokens[i].Print();
203     // }
204
205     return tokens;
206
207 }
208
209 vector<Token> LanguageDescriptorObject::Tokenize(Markup* input) {
210     vector<Token> tokens;
211
212     if (!input->IsLeaf()) {
213         vector<Markup*> children = input->Children();
214
215         for (int i = 0; i < children.size(); i++) {
216             vector<Token> tl = Tokenize(children[i]);
217             tokens.insert(tokens.end(), tl.begin(), tl.end());
218         }
219     } else {
220         Token t(input->GetID(), input->GetData());
221         tokens.push_back(t);
222     }
223
224     return tokens;
225
226 }
227
228 LanguageDescriptorObject::LanguageDescriptorObject()
229 {
230
231 }
232
233 LanguageDescriptorObject::LanguageDescriptorObject(string language)
234 {
235     Parse(language);
236 }
237
238 LanguageDescriptorObject::~LanguageDescriptorObject() {
239
240 }
241
242 void LanguageDescriptorObject::Parse(string language) {
243     // getpath function ?
244     string file = CFG_DIR + language + CFG_EXT;
245     FILE* temp = fopen((file).c_str(), "r");
246     if (temp != NULL) {
247         fclose(temp);
248     } else {
249         // try PATH environment variable?
250         throw "Cannot find language file.";
251     }
252     // return file;
253     //
254     this->language = language;
255
256     string data = Helpers::ReadFile(file); // TODO: file data should probably already be passed in?
257     string t = data;
258     ParseTerminalValues(data);
259     ParseFSM(data);
260     ParseReservedWords(data);
261     ParseIgnores(data);
262
263     regex r = regex("(.+?)\\s*=:\\s*([^]+?)\\n\\n");
264     smatch matches;
265
266     while (regex_search(t, matches, r)) {
267         Production* prod = new Production(this, matches[1], matches[2]);
```

```
268         productions.push_back(prod);
269         t = matches.suffix().str();
270     }
271 
272     // for (int i = 0; i < productions.size(); i++) {
273     //     cout << productions[i]->GetId() << ": " << productions[i]->GetRegex() << endl << endl;
274     // }
275 
276 }
277 
278 vector<Production*> LanguageDescriptorObject::GetProductions() {
279     return productions;
280 }
281 
282 Production* LanguageDescriptorObject::findProdById(string id) {
283     for (int i = 0; i < productions.size(); i++) {
284         if (productions[i]->GetId() == id) {
285             return productions[i];
286         }
287     }
288     return NULL;
289 }
290 int LanguageDescriptorObject::getProdIndex(string id) {
291     for (int i = 0; i < productions.size(); i++) {
292         if (productions[i]->GetId() == id) {
293             return i;
294         }
295     }
296     return -1;
297 }
298 
299 string LanguageDescriptorObject::GetLanguage() {
300     return language;
301 }
302 
303 vector<Production*> LanguageDescriptorObject::GetOrderedProductions(vector<string> stringlist) {
304     vector<Production*> v;
305 
306     int size = stringlist.size();
307     int* indexer = (int*)calloc(size, sizeof(int));
308     int i;
309     for (i = 0; i < size; i++) {
310         indexer[i] = getProdIndex(stringlist[i]);
311     }
312 
313     for (i = 1; i < size; i++) {
314         if (i > 0 && indexer[i - 1] < indexer[i]) {
315             string temps = stringlist[i];
316             stringlist[i] = stringlist[i - 1];
317             stringlist[i - 1] = temps;
318             int tempi = indexer[i];
319             indexer[i] =indexer[i - 1];
320             indexer[i - 1] = tempi;
321             i-=2;
322         }
323     }
324 
325     for (i = 0; i < size; i++) {
326         if (i == 0 || indexer[i] != indexer[i-1]) {
327             v.push_back(findProdById(stringlist[i]));
328         }
329     }
330 
331     return v;
332 }
333 
334 
```

```
335 Production::Production(LanguageDescriptorObject* ob, string id, string data) {
336     ldo = ob;
337     Parse(id, data);
338 }
339
340 TokenMatch* Production::Match(vector<Token> tokens) {
341     return Match(tokens, 0);
342 }
343
344 TokenMatch* Production::Match(vector<Token> tokens, int start) {
345     TokenMatch* t = rootSet->Match(tokens, start);
346     return t;
347 }
348
349 TokenMatch* Production::MatchStrict(vector<Token> tokens) {
350     return MatchStrict(tokens, 0);
351 }
352
353 TokenMatch* Production::MatchStrict(vector<Token> tokens, int start) {
354     TokenMatch* t = rootSet->MatchStrict(tokens, start);
355     return t;
356 }
357
358 void Production::Parse(string id, string data) {
359     this->id = id;
360     this->data = data;
361
362     rootSet = new ProductionSet(this);
363     rootSet->Parse(data);
364 }
365
366 string Production::GetRegex() {
367
368     vector<Production*> prods = GetContainedProductions();
369     string t = data;
370     regex r;
371     smatch matches;
372
373     for (int i = 0; i < prods.size(); i++) {
374         Production* prod = prods[i];
375         string sub = "(?:" + prod->GetRegex() + ")";
376         r = regex("<" + prod->GetId() + ">");
377
378         while (regex_search(t, matches, r)) {
379             t = matches.prefix().str() + sub + matches.suffix().str();
380         }
381     }
382
383     return t;
384 }
385
386 string Production::GetId() {
387     return id;
388 }
389
390 vector<Production*> Production::GetContainedProductions() {
391     vector<Production*> prods;
392     for (int i = 0; i < subproductions.size(); i++) {
393         Production* p = ldo->findProdById(subproductions[i]);
394         if (p != NULL) {
395             prods.push_back(p);
396         }
397     }
398     return prods;
399 }
400
401 LanguageDescriptorObject* Production::GetLDO() {
```

```
402     return ldo;
403 }
404
405 ProductionSet* Production::GetRootProductionSet() {
406     return rootSet;
407 }
408
409 ProductionSet::ProductionSet(Production* parentProduction) {
410     prod = parentProduction;
411 }
412
413 void ProductionSet::Parse(string data) {
414     source = data;
415
416     string a = "(?:\\$([^\\$]*?)\\$)"; // Action Routine
417     // string g = "(?:\\(([^\\)]*?)\\))"; // Group
418     string te = "(?:\\[(.*?)\\])"; // Terminal
419     string p = "(?:<(.*?)>)"; // Production
420     string m = "(\\?|\\*|\\+)"; // Multiplicity
421     string one = "(" + a + "|" + te + "|" + p + ")[\t ]*" + m + "?"; // One match
422     string alt = "(?:[ \t]*\\|[ \t]*)"; // Alternation sequence
423     string mult = "(?:" + alt + one + ")*"; // Multiple alternations
424     string reg;
425     if (type != _Alternation) {
426         reg = "(" + one + ")(" + mult + ")";
427     } else {
428         reg = "(" + one + ")()";
429     }
430
431     regex r = regex(reg);
432     smatch matches;
433     string t = data;
434
435     while (regex_search (t, matches, r)) {
436         ProductionSet* newSet = new ProductionSet(prod);
437
438         string actionRoutine = matches[3].str();
439         string terminal = matches[4].str();
440         string production = matches[5].str();
441         string multiplicity = matches[6].str();
442         string alternation = matches[7].str();
443
444         if (alternation == "") {
445             if (actionRoutine != "") {
446                 newSet->SetAction(actionRoutine);
447             } else if (terminal != "") {
448                 newSet->SetTerminal(terminal);
449             } else if (production != "") {
450                 newSet->SetProduction(production);
451             }
452             newSet->SetMultiplicity(multiplicity);
453         } else {
454             newSet->SetAlternation(matches[0]);
455         }
456         children.push_back(newSet);
457
458         t = matches.suffix().str();
459     }
460
461 }
462
463 void ProductionSet::SetAction(string data) {
464     type = _Action;
465     source = data;
466 }
467
468 void ProductionSet::SetTerminal(string data) {
```

```
469     type = _Terminal;
470     source = data;
471 }
472
473 void ProductionSet::SetProduction(string data) {
474     type = _Production;
475     source = data;
476 }
477
478 void ProductionSet::SetAlternation(string data) {
479     type = _Alternation;
480     Parse(data);
481 }
482
483 void ProductionSet::SetMultiplicity(string data) {
484     multiplicity = data;
485 }
486
487 TokenMatch* ProductionSet::Match(vector<Token> tokens) {
488     return Match(tokens, 0);
489 }
490
491 TokenMatch* ProductionSet::Match(vector<Token> tokens, int startIndex) {
492     TokenMatch* match;
493
494     for (int tokenIndex = startIndex; tokenIndex < tokens.size(); tokenIndex++) {
495         match = MatchStrict(tokens, tokenIndex);
496         if (match != NULL) {
497             return match;
498         }
499     }
500
501     return NULL;
502 }
503
504 Production* ProductionSet::GetProduction() {
505     return prod;
506 }
507
508 ProductionSetType ProductionSet::GetType() {
509     return type;
510 }
511
512 vector<ProductionSet*> ProductionSet::GetChildren() {
513     return children;
514 }
515
516 string ProductionSet::GetSource() {
517     return source;
518 }
519
520 string ProductionSet::GetMultiplicity() {
521     return multiplicity;
522 }
523
524
525
526 TokenMatch* ProductionSet::MatchStrict(vector<Token> tokens, int startIndex) {
527     TokenMatch* t = NULL;
528
529     if (type == _Terminal) {
530         t = MatchTerminal(tokens, startIndex);
531     }
532     else if (type == _Alternation) {
533         t = MatchAlternation(tokens, startIndex);
534     }
535     else if (type == _Group || type == _Root) {
```

```
536         t = MatchGroup(tokens, startIndex);
537         if (type == _Root && t != NULL) {
538             t->prod = GetProduction()->GetId();
539     //      cout << "Matched " << t->prod << endl;
540     //      cout << "Matched (" << source << "): count = " << t->length << ", start = " << t->begin << ", end = " << t->end << endl;
541     // for (int p = 0; p < t->match.size(); p++) {
542     //      cout << "\t" << t->match[p].id << endl;
543     // }
544     // cout << endl;
545         }
546     }
547     else if (type == _Production) {
548         t = MatchProduction(tokens, startIndex);
549     } else if (type == _Action) {
550         t = MatchAction(source, startIndex);
551     }
552 
553     return t;
554 }
555 TokenMatch* ProductionSet::MatchAction(string source, int startIndex) {
556 
557     TokenMatch* match = new TokenMatch();
558 
559     match->begin = startIndex;
560     match->end = startIndex;
561     match->length = 0;
562     match->isAction = true;
563     match->prod = source;
564 
565     return match;
566 }
567 
568 TokenMatch* ProductionSet::MatchGroup(vector<Token> tokens, int startIndex) {
569 
570     TokenMatch* match = new TokenMatch();
571     bool isMatch = true, matched = true;
572     int i = startIndex;
573 
574     TokenMatch* groupMatch;
575     for (int j = 0; j < children.size(); j++) {
576         groupMatch = children[j]->MatchStrict(tokens, i);
577         if (groupMatch == NULL) {
578             matched = false;
579             match->submatches.clear();
580             break;
581         }
582         if (groupMatch->length > 0 || groupMatch->isAction) {
583             match->submatches.push_back(groupMatch);
584             i += groupMatch->length;
585         }
586     }
587 
588     isMatch = multiplicity != "" || matched;
589 
590     if (!isMatch)
591         return NULL;
592 
593     match->begin = startIndex;
594     match->end = i;
595     match->length = match->end - match->begin;
596     match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);
597 
598     return match;
599 }
600 TokenMatch* ProductionSet::MatchTerminal(vector<Token> tokens, int startIndex) {
601 
602     if (startIndex >= tokens.size())
```

```
603         return NULL;
604 
605     TokenMatch* match = new TokenMatch();
606     bool isMatch = true, matched = false;
607 
608     matched = tokens[startIndex].id == source;
609     isMatch = multiplicity != "" || matched;
610 
611     if (!isMatch)
612         return NULL;
613 
614     match->begin = startIndex;
615     match->end = startIndex + (matched ? 1 : 0);
616     match->length = match->end - match->begin;
617     match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);
618 
619     return match;
620 }
621 TokenMatch* ProductionSet::MatchAlternation(vector<Token> tokens, int startIndex) {
622 
623     TokenMatch* match = new TokenMatch();
624     bool isMatch = true, matched = false;
625     int i = startIndex;
626 
627     TokenMatch* alternationMatch = NULL;
628     for (int j = 0; j < children.size(); j++) {
629         alternationMatch = children[j]->MatchStrict(tokens, i);
630         if (alternationMatch != NULL) {
631             matched = true;
632             if (alternationMatch->length > 0) {
633                 i += alternationMatch->length;
634                 match->submatches.push_back(alternationMatch);
635                 break;
636             }
637         }
638     }
639 
640     isMatch = multiplicity != "" || matched;
641 
642     if (!isMatch)
643         return NULL;
644 
645     match->begin = startIndex;
646     match->end = i;
647     match->length = match->end - match->begin;
648     match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);
649 
650     return match;
651 }
652 TokenMatch* ProductionSet::MatchProduction(vector<Token> tokens, int startIndex) {
653 
654     TokenMatch* match = new TokenMatch();
655     bool isMatch = true, matched = false;
656     int i = startIndex;
657 
658     Production* prod = this->prod->GetLDO()->findProdById(source);
659     if (prod != NULL) {
660         TokenMatch* prodMatch = prod->GetRootProductionSet()->MatchStrict(tokens, i);
661         if (prodMatch != NULL) {
662             if (prodMatch->length > 0) {
663                 i += prodMatch->length;
664                 match->submatches.push_back(prodMatch);
665             }
666             matched = true;
667         }
668     }
669 
```

```
670     isMatch = multiplicity != "" || matched;
671
672     if (!isMatch)
673         return NULL;
674
675     match->begin = startIndex;
676     match->end = i;
677     match->length = match->end - match->begin;
678     match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);
679
680     return match;
681 }
682
683 Markup* TokenMatch::GenerateMarkup(Markup* parent, bool addChildrenToParent) {
684     Markup* r = NULL;
685     if (addChildrenToParent) {
686         if (parent != NULL)
687             r = parent;
688         else
689             r = new Markup(prod);
690     } else {
691         r = new Markup(prod);
692         if (parent != NULL)
693             parent->AddChild(r);
694     }
695
696     string currentData;
697     vector<TokenMatch*> sms = submatches;
698
699     for (int i = 0; i <= length; i++) {
700         Markup* c = NULL;
701         TokenMatch* sub = NULL;
702
703         for (int j = 0; j < sms.size(); j++) {
704             if (sms[j]->begin == i + begin) {
705                 sub = sms[j];
706                 if (sub->isAction) {
707                     sms.erase(sms.begin() + j);
708                     break;
709                 }
710             }
711         }
712
713         if (sub != NULL) {
714             if (!sub->isAction) {
715                 c = sub->GenerateMarkup(r, sub->prod == "");
716                 i += sub->length - 1;
717             } else {
718                 ActionRoutines::ExecuteAction(sub->prod, r);
719                 i--;
720             }
721         } else if (i < length) {
722             c = new Markup(match[i].id, match[i].value);
723             r->AddChild(c);
724         }
725
726         if (c != NULL) {
727             if (currentData != "")
728                 currentData += " ";
729             currentData += c->GetData();
730         }
731     }
732     // r->SetData(currentData);
733
734     return r;
735 }
736
```

```
737 void TokenMatch::Print(int tab) {
738     if (prod != "") {
739         for (int p = 0; p < tab; p++)
740             cout << "\t";
741         cout << prod << endl;
742         tab++;
743     }
744 
745     for (int i = 0; i < length; i++) {
746         TokenMatch* sub = NULL;
747 
748         for (int j = 0; j < submatches.size(); j++) {
749             if (submatches[j]->begin == i + begin) {
750                 sub = submatches[j];
751                 break;
752             }
753         }
754 
755         if (sub != NULL) {
756             sub->Print(tab);
757             i += sub->length - 1;
758         } else {
759             for (int p = 0; p < tab; p++)
760                 cout << "\t";
761             cout << match[i].id << ": " << match[i].value << endl;
762         }
763     }
764 }
765 
766 unordered_map<string, ActionRoutine*> ActionRoutines::actions = {
767     { "DeclareVar", new DeclareVarAction() },
768     { "AssignVar", new AssignVarAction() },
769     { "ResolveExpr", new ResolveExprAction() },
770     { "AccumulateVar", new AccumulateVarAction() }
771 };
772 
773 Markup* ActionRoutines::ExecuteAction(string source, Markup* container) {
774     regex r = regex("^[ \t]*([a-zA-Z_][a-zA-Z_0-9]*)[ \t]*(?:\\((.*)\\))?[ \t]*$");
775     smatch matches;
776 
777     regex_search(source, matches, r);
778     string actionID = matches[1].str();
779     string actionParameters = matches[2].str();
780 
781     vector<Markup*> params = ResolveParameters(actionParameters, container);
782     return ExecuteAction(actionID, container, params);
783 }
784 Markup* ActionRoutines::ExecuteAction(string actionID, Markup* container, vector<Markup*> params) {
785 
786     ActionRoutine* action = NULL;
787     // cout << "Executed action " << actionID << endl;
788 
789     if ((action = ActionRoutines::actions[actionID]) != NULL) {
790         return action->Execute(container, params);
791     }
792 
793     return NULL;
794 }
795 vector<Markup*> ActionRoutines::ResolveParameters(string args, Markup* current) {
796     vector<Markup*> params;
797 
798     if (args != "") {
799         int groupLevel = 0;
800 
801         string arg = "";
802         for (int i = 0; i < args.size(); i++) {
803             if (args[i] == ',' && groupLevel == 0) {
```

```
804                 Markup* a = ResolveParameter(arg, current);
805                 params.push_back(a);
806                 arg = "";
807             } else {
808                 if (args[i] == '(')
809                     groupLevel++;
810                 else if (args[i] == ')')
811                     groupLevel--;
812                 arg += args[i];
813             }
814         }
815         if (arg != "") {
816             Markup* a = ResolveParameter(arg, current);
817             params.push_back(a);
818         }
819 
820     }
821 
822     return params;
823 }
824 Markup* ActionRoutines::ResolveParameter(string arg, Markup* current) {
825     regex fn = regex("^[ \t]*([a-zA-Z_][a-zA-Z_0-9]*)[ \t]*(\\(.*\\))?[ \t]*$");
826     smatch matches;
827 
828     // cout << "Arg: " << arg << endl;
829 
830     if (regex_search(arg, matches, fn)) {
831         string data = matches[0].str();
832         return ExecuteAction(data, current);
833     } else {
834 
835         int srcIndex = 0;
836         string subscript = "";
837         bool readSubscript = false;
838         bool readAncestor = false;
839 
840         regex indexReg = regex("^(\\+|\\-)?\\d+$");
841         regex sibOffsetReg = regex("^@((?:\\+|\\-)\\d+)$");
842         regex keyReg = regex("^(v)?\"(.*)\"$");
843         regex ancestorReg = regex("^\"(.*)\"$");
844 
845         for (int i = 0; i < arg.size() && current != NULL; i++) {
846             if (readSubscript) {
847                 if (arg[i] == ']') {
848                     readSubscript = false;
849                     if (regex_search(subscript, matches, indexReg)) {
850                         string index = matches[0].str();
851                         subscript = "";
852                         int n;
853                         istringstream(index) >> n;
854                         current = current->ChildAt(n);
855                         srcIndex = current->IndexInParent();
856                     } else if (regex_search(subscript, matches, keyReg)) {
857                         bool dive = matches[1].str() != "";
858                         string id = matches[2].str();
859                         subscript = "";
860                         if (dive)
861                             current = current->FindFirstById(id);
862                         else
863                             current = current->FindFirstChildById(id);
864                         srcIndex = current->IndexInParent();
865                     } else if (regex_search(subscript, matches, sibOffsetReg)) {
866                         string index = matches[1].str();
867                         subscript = "";
868                         int n;
869                         istringstream(index) >> n;
870                         n = srcIndex + n;
```

```
871                         current = current->ChildAt(n);
872                         srcIndex = current->IndexInParent();
873                     } else {
874                         cout << "Error parsing action routine parameter\n";
875                         subscript = "";
876                         break;
877                     }
878 
879                 } else {
880                     subscript += arg[i];
881                 }
882             } else if (readAncestor) {
883                 if (arg[i] == ')') {
884                     readAncestor = false;
885                     if (regex_search(subscript, matches, ancestorReg)) {
886                         string ancestor = matches[1].str();
887                         subscript = "";
888 
889                         int tempSrc;
890                         Markup* temp = current;
891                         do {
892                             tempSrc = temp->IndexInParent();
893                             temp = temp->Parent();
894                         } while (temp != NULL && temp->GetID() != ancestor);
895 
896                         if (temp != NULL) {
897                             srcIndex = tempSrc;
898                             current = temp;
899                         } else {
900                             cout << "Error parsing action routine parameter - Production '" << ancestor << "' not found as an ancestor to the current node.\n";
901                             break;
902                         }
903                     } else {
904                         cout << "Error parsing action routine parameter\n";
905                         subscript = "";
906                         break;
907                     }
908 
909                 } else {
910                     subscript += arg[i];
911                 }
912             } else {
913                 if (arg[i] == '^') {
914                     srcIndex = current->IndexInParent();
915                     current = current->Parent();
916                 } else if (arg[i] == '[') {
917                     readSubscript = true;
918                 } else if (arg[i] == '(') {
919                     readAncestor = true;
920                 }
921             }
922         }
923     }
924 
925     return current;
926 
927 }
928 Markup* DeclareVarAction::Execute(Markup* container, vector<Markup*> params) {
929     if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
930         // don't do anything with the expression for now
931         // this should be revised, but the incrementation screws with the Analyze module
932         return NULL;
933     }
934     if (params.size() >= 2 && params[0] != NULL && params[1] != NULL) {
935         string id = params[0]->GetData();
936         string type = params[1]->GetData();
937         Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
```

```
938         if (statement == NULL)
939             statement = container->FindAncestorById("function-definition");
940 
941         if (statement != NULL) {
942             statement->localDeclarations[id] = type;
943             cout << "Declared " << id << " with type " << type << endl;
944         }
945     } else {
946         cout << "Failed to read variable declaration\n";
947     }
948     return NULL;
949 }
950 Markup* AssignVarAction::Execute(Markup* container, vector<Markup*> params) {
951     if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
952         // don't do anything with the expression for now
953         // this should be revised, but the incrementation screws with the Analyze module
954         return NULL;
955     }
956     if (params.size() >= 2 && params[0] != NULL && params[1] != NULL) {
957         string id = params[0]->GetData();
958         Markup* value = params[1];
959         Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
960         if (statement == NULL)
961             statement = container->FindAncestorById("function-definition");
962 
963         if (statement != NULL) {
964             statement->localValues[id] = value;
965             cout << "Assigned " << id << " a value of " << value->GetData() << endl;
966         }
967     } else {
968         cout << "Failed to read assignment\n";
969     }
970     return NULL;
971 }
972 Markup* AccumulateVarAction::Execute(Markup* container, vector<Markup*> params) {
973     if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
974         // don't do anything with the expression for now
975         // this should be revised, but the incrementation screws with the Analyze module
976         return NULL;
977     }
978     if (params.size() >= 3 && params[0] != NULL && params[1] != NULL && params[2] != NULL) {
979         Markup* ident = params[1]->FindFirstById("identifier");
980         if (ident != NULL) {
981             string id = ident->GetData();
982             Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
983             if (statement == NULL)
984                 statement = container->FindAncestorById("function-definition");
985 
986             if (statement != NULL) {
987                 Markup* data = new Markup("algebraic-expression");
988                 data->localDeclarations = container->AccessibleDeclarations();
989                 data->localValues = container->AccessibleValues();
990 
991                 data->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { ident }));
992                 Markup* tail = new Markup("algebraic-expression-tail");
993                 Markup* expr = new Markup("operation-expression");
994                 string opVal = "";
995                 string assignOp = params[0]->GetID();
996                 string assignData = params[0]->GetData().substr(0, 1);
997                 if (assignOp == "PLUS_ASSIGN")
998                     opVal = "PLUS";
999                 else if (assignOp == "MINUS_ASSIGN")
1000                     opVal = "MINUS";
1001                 else if (assignOp == "ASTERISK_ASSIGN")
1002                     opVal = "ASTERISK";
1003                 else if (assignOp == "SLASH_ASSIGN")
1004                     opVal = "SLASH";
```

```
1005
1006                Markup* op = new Markup("math-binary-op");
1007                // TODO this won't work if the particular language doesn't have shorthand assignments like this
1008                op->AddChild(new Markup(opVal, assignData));
1009
1010                expr->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { params[2] }));
1011                tail->AddChild(op);
1012                tail->AddChild(expr);
1013                data->AddChild(tail);
1014                Markup* value = ActionRoutines::ExecuteAction("ResolveExpr", container, { data });
1015
1016                statement->localValues[id] = value;
1017                cout << "Assigned " << id << " a value of " << value->GetData() << endl;
1018            }
1019        }
1020    } else if(params.size() == 2 && params[0] != NULL && params[1] != NULL) {
1021        Markup* ident = params[1];
1022        Markup* uop = params[0]->ChildAt(0);
1023
1024        string id = ident->GetData();
1025        Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
1026        if (statement == NULL)
1027            statement = container->FindAncestorById("function-definition");
1028
1029        if (statement != NULL) {
1030            Markup* data = new Markup("algebraic-expression");
1031            data->localDeclarations = container->AccessibleDeclarations();
1032            data->localValues = container->AccessibleValues();
1033
1034            data->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { ident }));
1035            Markup* tail = new Markup("algebraic-expression-tail");
1036            Markup* expr = new Markup("operation-expression");
1037            Markup* op = NULL;
1038            if (uop->GetID() == "INCR") {
1039                op = new Markup("PLUS", "+");
1040            } else if (uop->GetID() == "DECR") {
1041                op = new Markup("MINUS", "-");
1042            }
1043            Markup* binaryOp = new Markup("math-binary-op");
1044            binaryOp->AddChild(op);
1045            tail->AddChild(binaryOp);
1046            expr->AddChild(new Markup("INT_LITERAL", "1"));
1047            tail->AddChild(expr);
1048            data->AddChild(tail);
1049            Markup* value = ActionRoutines::ExecuteAction("ResolveExpr", container, { data });
1050
1051            statement->localValues[id] = value;
1052            cout << "Assigned " << id << " a value of " << value->GetData() << endl;
1053        }
1054    } else {
1055        cout << "Failed to accumulate\n";
1056    }
1057    return NULL;
1058 }
1059 Markup* ResolveExprAction::Execute(Markup* container, vector<Markup*> params) {
1060    if (params.size() >= 1 && params[0] != NULL) {
1061        return ResolveExpr(params[0]);
1062    } else {
1063        cout << "Failed to resolve expression\n";
1064    }
1065    return NULL;
1066 }
1067 Markup* ResolveExprAction::ResolveExpr(Markup* data) {
1068    string id = data->GetID();
1069    // <grouped-expression>|<method-invocation>|<assignment>|<operation>|<simple-expression>
1070
1071    if (data->FindAncestorById("for-increment") != NULL || data->FindAncestorById("for-init") != NULL) {
```

```
1072        // don't do anything with the expression for now
1073        // this should be revised, but the incrementation screws with the Analyze module
1074    } else if (id == "assign-expression") {
1075        data = ResolveExpr(data->ChildAt(0));
1076    } if (id == "grouped-expression") {
1077        data = ResolveExpr(data->FindFirstChildById("expression")->ChildAt(0));
1078    } else if (id == "operation-expression") {
1079        data = ResolveExpr(data->ChildAt(0));
1080    } else if (id == "simple-expression") {
1081        data = ResolveExpr(data->ChildAt(0));
1082        // <member-access>|<subscript-access>|<literal>|<identifier>
1083        // TODO member-access & subscript-access?
1084    } else if (id == "literal") {
1085        data = data->ChildAt(0);
1086        // <bool-literal>|[FLOAT_LITERAL]|[INT_LITERAL]|[STRING_LITERAL]
1087    } else if (id == "identifier" || id == "ID") {
1088        unordered_map<string, Markup*> assignments = data->AccessibleValues();
1089        string var = data->GetData();
1090        if (assignments[var] != NULL) {
1091            data = assignments[var];
1092        } else {
1093            cout << "Variable " << var << " may be unassigned\n";
1094        }
1095    } else if (id == "operation") {
1096        data = ResolveExpr(data->ChildAt(0));
1097        //<binary-expression>|<unary-expression>
1098    } else if (id == "unary-expression") {
1099        // TODO
1100    } else if (id == "binary-expression") {
1101        data = ResolveExpr(data->ChildAt(0));
1102        //<relational-expression>|<algebraic-expression>|<logical-expression>
1103    } else if (id == "algebraic-expression") {
1104        vector<Markup*> operands = { ResolveExpr(data->ChildAt(0)) };
1105        vector<Markup*> operators;
1106        vector<Markup*> tails = data->FindFirstChildById("algebraic-expression-tail")->RecursiveElements();
1107        for (int i = 0; i < tails.size(); i++) {
1108            operators.push_back(tails[i]->FindFirstChildById("math-binary-op")->ChildAt(0));
1109            operands.push_back(ResolveExpr(tails[i]->FindFirstChildById("operation-expression")->ChildAt(0)));
1110        }
1111
1112        // Process multiplication and division
1113        for (int i = operators.size() - 1; i >= 0; i--) {
1114            Markup* op2 = operands[i + 1];
1115            Markup* op1 = operands[i];
1116            operands.erase(operands.begin() + i, operands.begin() + i + 2);
1117            Markup* op = operators[i];
1118            operators.erase(operators.begin() + i);
1119            string opId = op->GetID();
1120            // did both operands resolve to int literals
1121            if (op1->GetID() == "INT_LITERAL" && op2->GetID() == "INT_LITERAL" && (opId == "ASTERISK" || opId == "SLASH")) {
1122                long op1data, op2data, result;
1123                istringstream(op1->GetData()) >> op1data;
1124                istringstream(op2->GetData()) >> op2data;
1125                if (opId == "ASTERISK")
1126                    result = op1data * op2data;
1127                else if (opId == "SLASH")
1128                    result = op1data / op2data;
1129                Markup* r = new Markup("INT_LITERAL", to_string(result));
1130                operands.insert(operands.begin() + i, r);
1131            } else {
1132                operators.insert(operators.begin() + i, op);
1133                operands.insert(operands.begin() + i, op2);
1134                operands.insert(operands.begin() + i, op1);
1135            }
1136        }
1137        // process addition and subtraction
1138        for (int i = operators.size() - 1; i >= 0; i--) {
```

```
1139            Markup* op2 = operands[i + 1];
1140            Markup* op1 = operands[i];
1141            operands.erase(operands.begin() + i, operands.begin() + i + 2);
1142            Markup* op = operators[i];
1143            operators.erase(operators.begin() + i);
1144            string opId = op->GetID();
1145            // did both operands resolve to int literals?
1146            if (op1->GetID() == "INT_LITERAL" && op2->GetID() == "INT_LITERAL" && (opId == "PLUS" || opId == "MINUS")) {
1147                long op1data, op2data, result;
1148                istringstream(op1->GetData()) >> op1data;
1149                istringstream(op2->GetData()) >> op2data;
1150                if (opId == "MINUS")
1151                    op2data *= -1;
1152                result = op1data + op2data;
1153                Markup* r = new Markup("INT_LITERAL", to_string(result));
1154                operands.insert(operands.begin() + i, r);
1155            } else {
1156                operators.insert(operators.begin() + i, op->Clone());
1157                operands.insert(operands.begin() + i, op2->Clone());
1158                operands.insert(operands.begin() + i, op1->Clone());
1159            }
1160        }
1161
1162        if (operands.size() == 1) {
1163            data = operands[0];
1164        } else {
1165            data = new Markup("generated-expression");
1166            int i;
1167            for (i = 0; i < operators.size(); i++) {
1168                data->AddChild(operands[i]);
1169                data->AddChild(operators[i]);
1170            }
1171            data->AddChild(operands[i]);
1172        }
1173
1174    }
1175
1176    return data;
1177 }
```

```
/*
 *  LanguageDescriptor.h
 *  Defines the Language Descriptor class, which is the bridge between a text language descriptor file
 *
 *
 *  Created: 1/3/2017 by Ryan Tedeschi
 */

#ifndef LANGUAGE_DESCRIPTOR_H
#define LANGUAGE_DESCRIPTOR_H

#include <vector>
#include <string>
#include <sstream>
#include <regex>
#include <iostream>
#include <unordered_map>
#include "../Markup/Markup.h"
#include "../Helpers/Helpers.h"

#define CFG_EXT ".cfg"
#define CFG_DIR "./cfg/"

using namespace std;

enum ProductionSetType { _Root, _Terminal, _Group, _Alternation, _Production, _Action };

class Production;
class ProductionSet;
class LanguageDescriptorObject;
class TokenMatch;

class ActionRoutine {
    public:
        virtual Markup* Execute(Markup*, vector<Markup*>) = 0;
};

class DeclareVarAction : public ActionRoutine {
    public:
        Markup* Execute(Markup*, vector<Markup*>);
};
class AssignVarAction : public ActionRoutine {
    public:
        Markup* Execute(Markup*, vector<Markup*>);
};
class AccumulateVarAction : public ActionRoutine {
    public:
        Markup* Execute(Markup*, vector<Markup*>);
};
class ResolveExprAction : public ActionRoutine {
    public:
        Markup* Execute(Markup*, vector<Markup*>);

    private:
        Markup* ResolveExpr(Markup*);
};

class ActionRoutines {
    public:
        static Markup* ExecuteAction(string, Markup*);
        static Markup* ExecuteAction(string, Markup*, vector<Markup*>);

    private:
        static vector<Markup*> ResolveParameters(string, Markup*);
        static Markup* ResolveParameter(string, Markup*);

```

```
67          static unordered_map<string, ActionRoutine*> actions;
68  };
69
70  class Token : public Printable {
71      public:
72          Token(string, string);
73          string id;
74          string value;
75
76          void Print();
77
78      private:
79
80  };
81
82  class LanguageDescriptorObject
83  {
84      public:
85          LanguageDescriptorObject(string);
86          LanguageDescriptorObject();
87          ~LanguageDescriptorObject();
88
89          vector<Token> Tokenize(string);
90          vector<Token> Tokenize(Markup*);
91
92          void Parse(string);
93
94          string LookupTerminalValue(string);
95          bool IsTerminalIgnored(string);
96          Production* findProdById(string);
97          int getProdIndex(string);
98          vector<Production*> GetOrderedProductions(vector<string>);
99          vector<Production*> GetProductions();
100
101         string GetLanguage();
102
103
104     private:
105         void ParseTerminalValues(string);
106         void ParseFSM(string);
107         void ParseReservedWords(string);
108         void ParseIgnores(string);
109
110         unordered_map<string, bool> ignore;
111         unordered_map<string, string> terminals;
112         unordered_map<string, string> reservedWords;
113         vector<Production*> productions;
114         FSM<char> stateMachine;
115         string language;
116 };
117
118 class TokenMatch {
119     public:
120         bool isAction = false;
121         string prod;
122         int begin;
123         int end;
124         int length;
125         vector<Token> match;
126         vector<TokenMatch*> submatches;
127
128         Markup* GenerateMarkup(Markup* parent = NULL, bool addChildrenToParent = false);
129         void Print(int);
130
131     private:
132
133 };
```

```
134
135 class ProductionSet {
136     public:
137         ProductionSet(Production*);
138         void Parse(string);
139         void SetAction(string);
140         void SetTerminal(string);
141         void SetProduction(string);
142         void SetAlternation(string);
143         void SetMultiplicity(string);
144
145         TokenMatch* MatchStrict(vector<Token>, int);
146         TokenMatch* Match(vector<Token>, int);
147         TokenMatch* Match(vector<Token>);
148
149         Production* GetProduction();
150
151         ProductionSetType GetType();
152         vector<ProductionSet*> GetChildren();
153         string GetSource();
154         string GetMultiplicity();
155
156         // Markup Parser(vector<string>);
157
158     private:
159         TokenMatch* MatchGroup(vector<Token>, int);
160         TokenMatch* MatchTerminal(vector<Token>, int);
161         TokenMatch* MatchAlternation(vector<Token>, int);
162         TokenMatch* MatchProduction(vector<Token>, int);
163         TokenMatch* MatchAction(string, int);
164
165         Production* prod;
166         enum ProductionSetType type = _Root;
167         string source = "";
168         vector<ProductionSet*> children;
169         string multiplicity = "";
170 };
171
172 class Production {
173     public:
174         Production(LanguageDescriptorObject*, string, string);
175         void Parse(string, string);
176
177         LanguageDescriptorObject* GetLDO();
178         ProductionSet* GetRootProductionSet();
179         TokenMatch* Match(vector<Token>);
180         TokenMatch* Match(vector<Token>, int);
181         TokenMatch* MatchStrict(vector<Token>);
182         TokenMatch* MatchStrict(vector<Token>, int);
183
184         string GetRegex();
185         string GetId();
186         vector<Production*> GetContainedProductions();
187
188     private:
189         LanguageDescriptorObject* ldo;
190         string id;
191         string data;
192         vector<string> subproductions;
193         ProductionSet* rootSet;
194 };
195
196 #endif
```

```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 }
```

```
 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
```

```
 1 /*
 2  *  Printable.h
 3  *
 4  *
 5  *  Created: 4/2/2017 by Ryan Tedeschi
 6  */
 7
 8 #ifndef PRINTABLE_H
 9 #define PRINTABLE_H
10
11 class Printable {
12     virtual void Print() = 0;
13 };
14
15 #endif
```

```cpp
/*
 *  ControlModule.cpp
 *  Defines the Control Module class, which handles the flow of data between the consumer
 *      and the action modules
 *
 *  Created: 1/3/2017 by Ryan Tedeschi
 */

#include "ControlModule.h"

ControlModule::ControlModule() {

}

ControlModule::~ControlModule() {

}

void ControlModule::Run(SOURCE_LANGUAGE sourceLanguage, MODULE_ID moduleID, CODE_INPUT codeSnippets, FUNCTION_ARGS functionArgs) {

    LANGUAGE_DESCRIPTOR_OBJECT descriptor = NULL;
    CODE_OUTPUT code;
    MARKUP_OBJECT markup = NULL;
    bool cont = true;

    try {
        descriptor = GetLanguageDescriptor(sourceLanguage);
    } catch (...) {
        cont = false;
        returnData->AddStandardError("Language '" + sourceLanguage + "' could not be read. Could not proceed with execution.");
    }

    if (cont) {
        try {
            code = CoalesceCode(codeSnippets);
        } catch (...) {
            cont = false;
            returnData->AddStandardError("Error coalescing code. Could not proceed with execution");
        }
    }

    if (cont) {
        try {
            // TODO temporary passthrough
            code = codeSnippets;
            markup = Parse(code, descriptor);
        } catch (std::string message) {
            cont = false;
            returnData->AddStandardError("Error parsing code. Could not proceed with execution");
        }
    }

    if (cont)
        Execute(markup, descriptor, moduleID, functionArgs);

    FormatOutput();
}

LANGUAGE_DESCRIPTOR_OBJECT ControlModule::GetLanguageDescriptor(SOURCE_LANGUAGE sourceLanguage) throw (std::string) {
    return ReadLanguageFile(sourceLanguage);
}

bool ControlModule::ValidateSourceLanguage(SOURCE_LANGUAGE sourceLanguage) {
    return true;
}

```

```
67 LANGUAGE_DESCRIPTOR_OBJECT ControlModule::ReadLanguageFile(SOURCE_LANGUAGE sourceLanguage) throw (std::string) {
68     LANGUAGE_DESCRIPTOR_OBJECT languageDescriptor;
69     try {
70         // read and parse file;
71         languageDescriptor = new LanguageDescriptorObject(sourceLanguage);
72     } catch (...) {
73 
74         throw "Language '" + sourceLanguage + "' could not be read";
75     }
76 
77     return languageDescriptor;
78 }
79 
80 CODE_OUTPUT ControlModule::CoalesceCode(CODE_INPUT codeSnippets) {
81     CODE_OUTPUT code;
82 
83     // do some iterations over codeSnippets to unify it
84 
85     return code;
86 }
87 
88 MARKUP_OBJECT ControlModule::Parse(CODE_OUTPUT code, LANGUAGE_DESCRIPTOR_OBJECT languageDescriptor) {
89     MARKUP_OBJECT markup = new Markup("ROOT");
90 
91     vector<Token> tokens = languageDescriptor->Tokenize(code[0]);
92 
93     // for (int i = 0; i < tokens.size(); i++) {
94     //     cout << "State machine accepted token '" << tokens[i].id << "' with data '" << tokens[i].value << "'\n";
95     // }
96 
97     vector<Production*> prods = languageDescriptor->GetProductions();
98     vector<vector<Token>> tokenSets;
99     tokenSets.push_back(tokens);
100 
101     bool matched = false;
102 
103     for (int j = 0; j < tokenSets.size(); j++) {
104         matched = false;
105 
106         for (int p = 0; p < tokenSets[j].size() && !matched; p++) {
107 
108             for (int i = 0; i < prods.size() && !matched; i++) {
109                 TokenMatch* match = prods[i]->MatchStrict(tokenSets[j]);
110                 if (match != NULL) {
111                     vector<Token> s = vector<Token>(tokenSets[j].begin(), tokenSets[j].begin() + match->begin);
112                     vector<Token> e = vector<Token>(tokenSets[j].begin() + match->end, tokenSets[j].begin() + tokenSets[j].size());
113 
114                     tokenSets.erase(tokenSets.begin() + j);
115 
116                     bool dec = false;
117                     if (e.size() > 0) {
118                         tokenSets.insert(tokenSets.begin() + j, e);
119                         dec = true;
120                     }
121                     if (s.size() > 0) {
122                         tokenSets.insert(tokenSets.begin() + j, s);
123                         dec = true;
124                     }
125                     if (dec)
126                         j--;
127                     // cout << "MATCHED: " << prods[i]->GetId() << ", start = " << match->begin << ", end = " << match->end << ", length = " << match->length << endl;
128                     // match->Print(0);
129                     Markup* m = match->GenerateMarkup();
130                     markup->AddChild(m);
131                     // markupList.push_back(m);
132                     matched = true;
133                 }
```

```
134             }
135         }
136     }
137
138     // markup->Print();
139
140     return markup;
141 }
142
143 void ControlModule::Execute(MARKUP_OBJECT markup, LANGUAGE_DESCRIPTOR_OBJECT ldo, MODULE_ID moduleID, FUNCTION_ARGS functionArgs) {
144     MODULE_REF ref = ModuleRetrieval(moduleID);
145     if (ref != NULL)
146         ModuleExecution(ref, markup, ldo, functionArgs);
147 }
148
149 MODULE_REF ControlModule::ModuleRetrieval(MODULE_ID moduleID) {
150
151     try {
152         return GetModule(moduleID);
153     } catch (...) {
154         returnData->AddStandardError("Module '" + moduleID + "' could not be found.");
155         return NULL;
156     }
157
158 }
159
160 void ControlModule::ModuleExecution(MODULE_REF moduleRef, MARKUP_OBJECT markup, LANGUAGE_DESCRIPTOR_OBJECT ldo, FUNCTION_ARGS functionArgs) {
161     try {
162         // attempt to execute the module
163         returnData = moduleRef->Execute(markup, ldo, functionArgs, returnData);
164     } catch (...) {
165         returnData->AddStandardError("An error occurred while trying to execute the module!");
166     }
167 }
168
169 void ControlModule::FormatOutput() {
170
171     cout << "CASP_RETURN_DATA_START\n";
172     returnData->Print();
173     cout << "\nCASP_RETURN_DATA_END\n";
174
175 }
```

```
/*
 *  ControlModule.h
 *  Defines the Control Module class, which handles the flow of data between the consumer
 *      and the action modules
 *
 *  Created: 1/3/2017 by Ryan Tedeschi
 */

#ifndef CONTROLMODULE_H
#define CONTROLMODULE_H

/*
 *  Control Module Input parameter type placeholders
 */
#define SOURCE_LANGUAGE string
#define MODULE_ID string
#define CODE_INPUT vector<string>
#define FUNCTION_ARGS vector<arg>
#define MODULE_RESPONSE CASP_Return*
#define LANGUAGE_DESCRIPTOR_OBJECT LanguageDescriptorObject*
#define MARKUP_OBJECT Markup*
#define CODE_OUTPUT vector<string>
#define MODULE_REF CASP_Plugin*

#include "../shared/CASP_Plugin/CASP_Plugin.h"
#include "../plugins/plugins.h"
#include <string>
#include "../shared/LanguageDescriptor/LanguageDescriptor.h"
#include "../shared/Markup/Markup.h"

using namespace std;

class ControlModule {
    public:
        ControlModule();
        ~ControlModule();
        void Run(SOURCE_LANGUAGE, MODULE_ID, CODE_INPUT, FUNCTION_ARGS);

    private:
        LANGUAGE_DESCRIPTOR_OBJECT GetLanguageDescriptor(SOURCE_LANGUAGE) throw (std::string);
        bool ValidateSourceLanguage(SOURCE_LANGUAGE);
        LANGUAGE_DESCRIPTOR_OBJECT ReadLanguageFile(SOURCE_LANGUAGE) throw (std::string);
        CODE_OUTPUT CoalesceCode(CODE_INPUT); // is this necessary?..
        MARKUP_OBJECT Parse(CODE_OUTPUT, LANGUAGE_DESCRIPTOR_OBJECT);
        void Execute(MARKUP_OBJECT, LANGUAGE_DESCRIPTOR_OBJECT, MODULE_ID, FUNCTION_ARGS);
        MODULE_REF ModuleRetrieval(MODULE_ID);
        void ModuleExecution(MODULE_REF, MARKUP_OBJECT, LANGUAGE_DESCRIPTOR_OBJECT, FUNCTION_ARGS);
        void FormatOutput();

        CASP_Return* returnData = new CASP_Return();
};

#endif
```

```cpp
/*
 *  Main.cpp
 *  Initiates the program
 *
 *  Created: 1/3/2017 by Ryan Tedeschi
 */

#include "../shared/Helpers/Helpers.h"
#include "ControlModule.h"
#include "Timestamp.h"
#include <iostream>
#include <string>
#include <list>

#define OTHER 0
#define MODULE 1
#define LANGUAGE 2
#define CODE 3
#define CODE_F 4
int SIZES[] { 0, 10, 12, 6, 7 };

int paramType(string);
string parseParam(string, int);
bool stripArgData(string, string*, string*);
string unescapeCharacters(string);

int main(int argCount, char** argArray)
{
    cout << "Code Analyzer Software Package (CASP)" << endl;
    cout << "Build Date: " << TIMESTAMP << endl;

    vector<arg> fnArgs;
    vector<string> codeSource;
    string sourceLanguage;
    string moduleID;

    for (int i = 1; i < argCount; i++)
    {
        int type = paramType(argArray[i]);
        string value = parseParam(argArray[i], type);
        switch (type) {
            case MODULE:
                moduleID = value;
                break;
            case LANGUAGE:
                sourceLanguage = value;
                break;
            case CODE_F:
                value = Helpers::ReadFile(value);
            case CODE:
                codeSource.push_back(value);
                break;
            default:
                string t, v;
                if (stripArgData(argArray[i], &t, &v)) {
                    arg a(t, v);
                    fnArgs.push_back(a);
                }
        }
    }

    ControlModule control = ControlModule();
    control.Run(sourceLanguage, moduleID, codeSource, fnArgs);

    cout << "CASP - Operation Complete";

```

```
67     return 0;
68 }
69
70 int paramType(string input) {
71     int type = OTHER;
72     if (input.find("/sourcelang=") == 0)
73         type = LANGUAGE;
74     else if (input.find("/moduleid=") == 0)
75         type = MODULE;
76     else if (input.find("/code=") == 0)
77         type = CODE;
78     else if (input.find("/codef=") == 0)
79         type = CODE_F;
80     return type;
81 }
82
83 bool stripArgData(string input, string* id, string* value) {
84     int slash = input.find("/");
85     int eq = input.find("=");
86
87     if (slash != 0) {
88         *id = "";
89         *value = "";
90         return false;
91     }
92
93     if (eq == -1) {
94         *id = Helpers::toLower(input.substr(1, input.size()));
95         *value = "";
96         return true;
97     }
98
99     *id = Helpers::toLower(input.substr(1, eq - 1));
100    *value = input.substr(eq + 1, input.size());
101    return true;
102 }
103
104 string parseParam(string input, int type)
105 {
106    int size = SIZES[type];
107    string out = input.substr(size, input.size() - size);
108    return out;
109 }
```

```
1 /*
2 * It is not advised to edit this file, as it
3 * is automatically generated with every build
4 */
5 #ifndef TIMESTAMP
6 #define TIMESTAMP "04-28-2017 14:12:38"
7 #endif
```

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.ComponentModel;
 4 using System.Data;
 5 using System.Drawing;
 6 using System.Linq;
 7 using System.Text;
 8 using System.Threading.Tasks;
 9 using System.Windows.Forms;
10
11 namespace CASP_Standalone_Implementation.Forms
12 {
13     public partial class AddArgForm : Form
14     {
15         private Action<string, string> Callback;
16
17         public AddArgForm(Action<string, string> callback)
18         {
19             Callback = callback;
20             InitializeComponent();
21         }
22
23         private void AddButton_Click(object sender, EventArgs e)
24         {
25             Callback(ArgNameTextbox.Text, ArgValueTextbox.Text);
26             Close();
27             Dispose();
28         }
29
30         private void CancelButton_Click(object sender, EventArgs e)
31         {
32             Close();
33             Dispose();
34         }
35     }
36 }
```

```
using System.Drawing;
using System.Windows.Forms;
using CASP_Standalone_Implementation.Src;
using Newtonsoft.Json.Linq;
using System.Collections.Generic;
using System;
using System.Linq;
using System.Drawing.Drawing2D;

namespace CASP_Standalone_Implementation.Forms
{
    public partial class CASP_AnalyzeForm: CASP_OutputForm // Form
    {

        public CASP_AnalyzeForm()
        {
            InitializeComponent();
        }

        public override void Set_CASP_Output(JObject CASP_Response)
        {
            ParseResponse(CASP_Response);
        }

        private void ParseResponse(JObject CASP_Response)
        {
            JObject data = (JObject)CASP_Response["Data"];

            int y = fnlabel.Height + 5;

            foreach (KeyValuePair<string, JToken> prop in data)
            {
                JObject ob = (JObject)prop.Value;
                bool undefined = (bool)ob["IsUndefined"];
                string title = prop.Key;
                string analysis = (string)ob["Analysis"];

                Label Title = new Label();
                Title.AutoSize = false;
                Title.Font = fnlabel.Font;
                Title.Size = fnlabel.Size;
                Point TitleLocation = fnlabel.Location;
                TitleLocation.Y = y;
                Title.Location = TitleLocation;
                Title.TextAlign = ContentAlignment.MiddleCenter;
                Title.Text = title;

                Label Analysis = new Label();
                Analysis.AutoSize = false;
                Analysis.Font = fnlabel.Font;
                Analysis.Size = fnlabel.Size;
                Point AnalysisLocation = complexitylabel.Location;
                AnalysisLocation.Y = y;
                Analysis.Location = AnalysisLocation;
                Analysis.TextAlign = ContentAlignment.MiddleCenter;
                Analysis.Text = analysis;

                y += fnlabel.Height + 5;

                Parent.Controls.Add(Title);
                Parent.Controls.Add(Analysis);

                //AnalysisTable.Items.Add(title + ":\t" + analysis);
            }

        }
```

```
67
68          private void AnalysisTable_Paint(object sender, PaintEventArgs e)
69          {
70
71          }
72      }
73  }
```

```
using System.Drawing;
using System.Windows.Forms;
using CASP_Standalone_Implementation.Src;
using Newtonsoft.Json.Linq;
using System.Collections.Generic;
using System;
using System.Linq;
using System.Drawing.Drawing2D;

namespace CASP_Standalone_Implementation.Forms
{
    public partial class CASP_OutlineForm : CASP_OutputForm // Form
    {
        Pen blockPen = Pens.Gray;

        public CASP_OutlineForm()
        {
            InitializeComponent();
        }

        public override void Set_CASP_Output(JObject CASP_Response)
        {
            List<OutlineGraph> graphs = ParseResponse(CASP_Response);
            int x = 0;

            for (int i = 0; i < graphs.Count; i++)
            {
                Panel p = BreadthFirstDraw(graphs[i]);
                //p.BorderStyle = BorderStyle.FixedSingle;
                p.Location = new Point(x, 0);
                FlowPanel.Controls.Add(p);

                x += p.Width + 20;
                graphs[i].Reset();
            }

        }

        private FlowBlock DrawNode(OutlineNode node, FlowBlock parent, Panel panel, int minX, int y, out Point newPoint)
        {

            node.drawn = true;
            FlowBlock block = GetFlowBlock(node);
            panel.Controls.Add(block);
            int preferredX;

            if (parent != null)
            {
                parent.children.Add(block);
                block.parent = parent;
                preferredX = parent.Center.X - block.Width / 2;
            }
            else
            {
                preferredX = minX;
            }

            block.Location = new Point(Math.Max(minX, preferredX), y);

            newPoint = new Point(block.Right, block.Bottom);
            return block;
        }

        private class node
        {
            public FlowBlock parentFlow;
```

```
67              public List<OutlineNode> children;
68          }
69
70          private Panel BreadthFirstDraw(OutlineGraph graph)
71          {
72              OutlineNode head = graph.nodes[0];
73
74              Panel panel = new Panel();
75              panel.AutoSize = true;
76              int yBuff = 30;
77              int xBuff = 30;
78
79              int y = yBuff;
80              List<FlowBlock> blocks = new List<FlowBlock>();
81              Dictionary<int, FlowBlock> blockDictionary = new Dictionary<int, FlowBlock>();
82              List<List<node>> levels = new List<List<node>>() { new List<node>() { new node() { parentFlow = null, children = new List<OutlineNode>() { head } } } };
83              for (int i = 0; i < levels.Count; i++)
84              {
85
86                  List<node> nodes = levels[i];
87                  int levelY = y;
88                  int minX = xBuff;
89                  for (int k = 0; k < nodes.Count; k++)
90                  {
91                      List<OutlineNode> n = nodes[k].children;
92                      FlowBlock parent = nodes[k].parentFlow;
93
94                      for (int j = 0; j < n.Count; j++)
95                      {
96                          OutlineNode node = n[j];
97                          Point newCoords;
98
99                          FlowBlock block = DrawNode(node, parent, panel, minX, levelY, out newCoords);
100
101                         // TODO need to work on decisions
102                         if (blockDictionary.ContainsKey(node.index))
103                         {
104                             FlowBlock old = blockDictionary[node.index];
105                             blockDictionary.Remove(node.index);
106                             blocks.Remove(old);
107                             panel.Controls.Remove(old);
108                         }
109                         else
110                         {
111                             node newNode = new node
112                             {
113                                 parentFlow = block,
114                                 children = node.edges
115                                 .Where(e => !e.target.drawn)
116                                 .Select(e => e.target)
117                                 .ToList()
118                             };
119
120                             if (levels.Count > i + 1)
121                                 levels[i + 1].Add(newNode);
122                             else
123                                 levels.Add(new List<node>() { newNode });
124                         }
125
126                         blockDictionary.Add(node.index, block);
127                         blocks.Add(block);
128
129                         minX = block.Right + xBuff;
130                         if (newCoords.Y > y)
131                             y = newCoords.Y;
132
133                     }
```

```
134                 }
135 
136                 y += yBuff;
137             }
138 
139             for (int i = 0; i < graph.edges.Count; i++)
140             {
141                 OutlineEdge edge = graph.edges[i];
142                 FlowBlock source = blockDictionary[edge.source.index];
143                 FlowBlock target = blockDictionary[edge.target.index];
144 
145                 if (!source.ConnectTo(target, edge.text))
146                 {
147                     // uh-oh... not enough space on the node. Should only happen on switch, which we don't have
148                 }
149             }
150 
151             RenderEdges(blocks, panel);
152 
153             return panel;
154         }
155 
156         private void RenderEdges(List<FlowBlock> blocks, Panel panel)
157         {
158             panel.Paint += (object sender, PaintEventArgs e) =>
159             {
160                 for (int i = 0; i < blocks.Count; i++)
161                     blocks[i].RenderEdgeGraphics(e.Graphics);
162             };
163         }
164 
165         FlowBlock GetFlowBlock(OutlineNode node)
166         {
167             FlowBlock block = null;
168             switch (node.type)
169             {
170                 case BlockType.Decision:
171                     block = GetFlowDecision(node.text);
172                     break;
173                 case BlockType.End:
174                     block = GetFlowEnd(node.text);
175                     break;
176                 case BlockType.EndDecision:
177                     block = GetFlowSink(node.text);
178                     break;
179                 //case BlockType.IO:
180                 //    block = GetFlowDecision(node.text);
181                 //    break;
182                 case BlockType.Loop:
183                     block = GetFlowLoop(node.text);
184                     break;
185                 case BlockType.MethodCall:
186                     block = GetFlowMethod(node.text);
187                     break;
188                 case BlockType.Process:
189                     block = GetFlowProcess(node.text);
190                     break;
191                 case BlockType.Start:
192                     block = GetFlowEnd(node.text);
193                     break;
194                 default:
195                     block = new FlowBlock();
196                     break;
197             }
198             block.UpdateSockets();
199             block.id = node.index;
200             block.type = node.type;
```

```
201
202             return block;
203         }
204
205         private List<OutlineGraph> ParseResponse(JObject CASP_Response)
206         {
207             JObject data = (JObject)CASP_Response["Data"];
208             JArray outlines = (JArray)data["Outlines"];
209
210             List<OutlineGraph> graphs = new List<OutlineGraph>();
211
212             if (outlines != null)
213             {
214
215                 for (int i = 0; i < outlines.Count; i++)
216                 {
217                     List<dynamic> edgeList = new List<dynamic>();
218                     JArray o = (JArray)outlines[i];
219
220                     OutlineGraph graph = new OutlineGraph();
221
222                     for (int j = 0; j < o.Count; j++)
223                     {
224                         JObject node = (JObject)o[j];
225                         string nodeText = (string)node["data"];
226                         BlockType nodeType = (BlockType)Enum.Parse(typeof(BlockType), (string)node["type"]);
227                         JArray edges = (JArray)node["edges"];
228
229                         graph.AddNode(new OutlineNode() { text = nodeText, type = nodeType });
230
231                         for (int k = 0; k < edges.Count; k++)
232                         {
233                             JObject edge = (JObject)edges[k];
234                             int source = (int)edge["source"];
235                             int target = (int)edge["target"];
236                             string edgeText = (string)edge["data"];
237
238                             edgeList.Add(new { source = source, target = target, text = edgeText });
239                         }
240                     }
241
242                     for (int j = 0; j < edgeList.Count; j++)
243                     {
244                         graph.AddEdge(edgeList[j].source, edgeList[j].target, edgeList[j].text);
245                     }
246
247                     graphs.Add(graph);
248                 }
249
250                 if (graphs.Count == 0)
251                 {
252                     OutlineGraph graph = new OutlineGraph();
253                     graph.AddNode(new OutlineNode()
254                     {
255                         index = 0,
256                         text = "No flowchart data\nto display!",
257                         type = BlockType.Process
258                     });
259                     graphs.Add(graph);
260                 }
261             }
262
263             return graphs;
264         }
265
266         FlowBlock GetFlowEnd(string text)
267         {
```

```
268             FlowBlock flowblock = CreateFlowblock(text);
269             flowblock.Paint += PaintFlowblockEnd;
270             return flowblock;
271         }
272 
273         FlowBlock GetFlowProcess(string text)
274         {
275             FlowBlock flowblock = CreateFlowblock(text);
276             flowblock.Paint += PaintFlowblockProcess;
277             return flowblock;
278         }
279 
280         FlowBlock GetFlowDecision(string text)
281         {
282             FlowBlock flowblock = CreateFlowblock(text);
283             flowblock.Paint += PaintFlowblockDecision;
284             return flowblock;
285         }
286 
287         FlowBlock GetFlowSink(string text)
288         {
289             FlowBlock flowblock = CreateFlowblock("");
290             flowblock.Width = flowblock.Height = 53;
291             flowblock.Paint += PaintFlowblockSink;
292             return flowblock;
293         }
294 
295         FlowBlock GetFlowMethod(string text)
296         {
297             FlowBlock flowblock = CreateFlowblock(text);
298             flowblock.Paint += PaintFlowblockMethod;
299             return flowblock;
300         }
301 
302         FlowBlock GetFlowLoop(string text)
303         {
304             FlowBlock flowblock = CreateFlowblock(text);
305             flowblock.Paint += PaintFlowblockLoop;
306             return flowblock;
307         }
308 
309         private void PaintFlowblockEnd(object sender, PaintEventArgs e)
310         {
311             int left, right, top, bottom, centerX, centerY;
312             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
313 
314             Graphics g = e.Graphics;
315             g.DrawArc(blockPen, new Rectangle(left - 5, top, 10, bottom - top), 90, 180);
316             g.DrawLine(blockPen, left, top, right, top);
317             g.DrawArc(blockPen, new Rectangle(right - 5, top, 10, bottom - top), -90, 180);
318             g.DrawLine(blockPen, right, bottom, left, bottom);
319         }
320 
321         private void PaintFlowblockProcess(object sender, PaintEventArgs e)
322         {
323             int left, right, top, bottom, centerX, centerY;
324             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
325 
326             Graphics g = e.Graphics;
327             g.DrawLine(blockPen, left, top, right, top);
328             g.DrawLine(blockPen, right, top, right, bottom);
329             g.DrawLine(blockPen, right, bottom, left, bottom);
330             g.DrawLine(blockPen, left, bottom, left, top);
331         }
332 
333         private void PaintFlowblockDecision(object sender, PaintEventArgs e)
334         {
```

```
335             int left, right, top, bottom, centerX, centerY;
336             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
337 
338             Graphics g = e.Graphics;
339             g.DrawLine(blockPen, centerX, top - 8, right + 8, centerY);
340             g.DrawLine(blockPen, right + 8, centerY, centerX, bottom + 8);
341             g.DrawLine(blockPen, centerX, bottom + 8, left - 8, centerY);
342             g.DrawLine(blockPen, left - 8, centerY, centerX, top - 8);
343         }
344 
345         private void PaintFlowblockMethod(object sender, PaintEventArgs e)
346         {
347             int left, right, top, bottom, centerX, centerY;
348             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
349 
350             Graphics g = e.Graphics;
351             g.DrawLine(blockPen, left - 5, top, right + 5, top);
352             g.DrawLine(blockPen, right + 5, top, right + 5, bottom);
353             g.DrawLine(blockPen, right, top, right, bottom);
354             g.DrawLine(blockPen, right + 5, bottom, left - 5, bottom);
355             g.DrawLine(blockPen, left - 5, bottom, left - 5, top);
356             g.DrawLine(blockPen, left, bottom, left, top);
357         }
358 
359         private void PaintFlowblockLoop(object sender, PaintEventArgs e)
360         {
361             int left, right, top, bottom, centerX, centerY;
362             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
363 
364             Graphics g = e.Graphics;
365             g.DrawLine(blockPen, left + 5, top, right - 5, top);
366             g.DrawLine(blockPen, right - 5, top, right + 5, centerY);
367             g.DrawLine(blockPen, right + 5, centerY, right - 5, bottom);
368             g.DrawLine(blockPen, right - 5, bottom, left + 5, bottom);
369             g.DrawLine(blockPen, left + 5, bottom, left - 5, centerY);
370             g.DrawLine(blockPen, left - 5, centerY, left + 5, top);
371         }
372 
373         private void PaintFlowblockSink(object sender, PaintEventArgs e)
374         {
375             int left, right, top, bottom, centerX, centerY;
376             FlowBlock flowblock = ReadFlowblockData(sender, out left, out right, out top, out bottom, out centerX, out centerY);
377 
378             Graphics g = e.Graphics;
379             g.DrawEllipse(blockPen, new Rectangle(left, top, right - left, bottom - top));
380         }
381 
382 
383         private FlowBlock CreateFlowblock(string text)
384         {
385             FlowBlock flowblock = new FlowBlock();
386             flowblock.Text = text;
387             flowblock.AutoSize = true;
388 
389             FlowPanel.Controls.Add(flowblock);
390             int width = flowblock.Width;
391             int height = flowblock.Height;
392             FlowPanel.Controls.Remove(flowblock);
393 
394             flowblock.BackColor = Color.Transparent;
395             flowblock.TextAlign = ContentAlignment.MiddleCenter;
396             flowblock.AutoSize = false;
397             flowblock.Width = width + 40;
398             flowblock.Height = height + 40;
399 
400             flowblock.Cursor = Cursors.Hand;
401 
```

```
402             //flowblock.BorderStyle = BorderStyle.Fixed3D;
403 
404             return flowblock;
405         }
406 
407         private FlowBlock ReadFlowblockData(object sender, out int left, out int right, out int top, out int bottom, out int centerX, out int centerY)
408         {
409             FlowBlock flowblock = sender as FlowBlock;
410 
411             int width = flowblock.Width;
412             int height = flowblock.Height;
413 
414             left = 10;
415             right = width - 10 - 2;
416             top = 10;
417             bottom = height - 10 - 2;
418             centerX = left + (right - left) / 2;
419             centerY = top + (bottom - top) / 2;
420 
421             return flowblock;
422         }
423     }
424 }
```

```
1 using System.Drawing;
2 using System.Windows.Forms;
3 using CASP_Standalone_Implementation.Src;
4 using Newtonsoft.Json.Linq;
5 using System.Collections.Generic;
6 using System;
7 using System.Linq;
8 using System.Drawing.Drawing2D;
9
10 namespace CASP_Standalone_Implementation.Forms
11 {
12     public partial class CASP_PrintForm : CASP_OutputForm
13     {
14         class Tree
15         {
16             public string Title;
17             public string Data;
18
19             public List<Tree> Children = new List<Tree>();
20
21             public bool isLeaf
22             {
23                 get
24                 {
25                     return Children.Count == 0;
26                 }
27             }
28         }
29
30         public CASP_PrintForm()
31         {
32             InitializeComponent();
33         }
34
35         public override void Set_CASP_Output(JObject CASP_Response)
36         {
37             Tree t = Parse((JObject)CASP_Response["Data"]["ParseTree"]);
38
39             TreeView.Nodes.Add(Display(t));
40
41             Display(t);
42         }
43
44         TreeNode Display(Tree t)
45         {
46             TreeNode node = new TreeNode();
47             node.Text = t.Title;
48
49             if (!t.isLeaf)
50             {
51                 for (int i = 0; i < t.Children.Count; i++)
52                     node.Nodes.Add(Display(t.Children[i]));
53             }
54             else
55             {
56                 node.Nodes.Add(new TreeNode(t.Data));
57             }
58
59             node.Expand();
60
61             return node;
62         }
63
64         Tree Parse(JObject response)
65         {
66             Tree newTree = new Tree();
```

```
67              JArray children = (JArray)response["Children"];
68
69              newTree.Title = (string)response["Title"];
70              newTree.Data = (string)response["Data"];
71
72              for (int i = 0; i < children.Count; i++)
73              {
74                  newTree.Children.Add(Parse((JObject)children[i]));
75              }
76
77              return newTree;
78          }
79
80      }
81  }
```

```
using System.Drawing;
using System.Windows.Forms;
using CASP_Standalone_Implementation.Src;
using Newtonsoft.Json.Linq;
using System.Collections.Generic;
using System;
using System.Linq;
using System.Drawing.Drawing2D;

namespace CASP_Standalone_Implementation.Forms
{
    public partial class CASP_TranslateForm : CASP_OutputForm // Form
    {
        class Output
        {
            public KeyValuePair<string, string> Source { get; set; }
            public KeyValuePair<string, string> Target { get; set; }
        }

        public CASP_TranslateForm()
        {
            InitializeComponent();
        }

        public override void Set_CASP_Output(JObject CASP_Response)
        {
            Output output = ParseResponse(CASP_Response);

            SourceLanguageBox.Text = output.Source.Key;
            SourceDataBox.Text = output.Source.Value;
            TargetLanguageBox.Text = output.Target.Key;
            TargetDataBox.Text = output.Target.Value;

        }

        private Output ParseResponse(JObject CASP_Response)
        {
            JObject data = (JObject)CASP_Response["Data"];
            JObject source = (JObject)data["OriginalSource"];
            JObject target = (JObject)data["TranslatedSource"];

            string slanguage = "", sdata = "", tlanguage = "", tdata = "";

            if (source != null)
            {
                slanguage = source["Language"].ToString();
                sdata = source["Data"].ToString();
            }
            if (target != null)
            {
                tlanguage = target["Language"].ToString();
                tdata = target["Data"].ToString();
            }

            return new Output() {
                Source = new KeyValuePair<string, string>(slanguage, sdata),
                Target = new KeyValuePair<string, string>(tlanguage, tdata)
            };
        }

    }
}
```

```
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace CASP_Standalone_Implementation.Forms
{
    public partial class ErrorProviderForm : Form
    {
        int numErrors;
        int numWarnings;

        public int NumErrors
        {
            get
            {
                return numErrors;
            }
        }
        public int NumWarnings
        {
            get
            {
                return numWarnings;
            }
        }

        public ErrorProviderForm(JObject CASP_Response)
        {
            InitializeComponent();
            ShowErrors(CASP_Response);
            ShowWarnings(CASP_Response);
        }

        void ShowErrors(JObject CASP_Response)
        {
            JArray errors = (JArray)CASP_Response["Errors"];
            ShowInList(errors, ErrorList);
            numErrors = errors.Count;
            ErrorsTab.Text = "Errors (" + numErrors + ")";
        }

        void ShowWarnings(JObject CASP_Response)
        {
            JArray warnings = (JArray)CASP_Response["Warnings"];
            ShowInList(warnings, WarningList);
            numWarnings = warnings.Count;
            WarningsTab.Text = "Warnings (" + numWarnings + ")";
        }

        void ShowInList(JArray list, ListBox control)
        {
            for (int i = 0; i < list.Count; i++)
                control.Items.Add((string)list[i]["message"]);
        }

    }
}
```

```
 1 using System;
 2 using System.Data;
 3 using System.IO;
 4 using System.Linq;
 5 using System.Windows.Forms;
 6 using CASP_Standalone_Implementation.Forms;
 7 using CASP_Standalone_Implementation.Src;
 8 using System.Collections.Generic;
 9 using System.Text.RegularExpressions;
10 using Newtonsoft.Json;
11 using Newtonsoft.Json.Linq;
12
13 namespace CASP_Standalone_Implementation
14 {
15     public partial class MainForm : Form
16     {
17         public static string[] Languages
18         {
19             get
20             {
21                 string dir = ConsoleWrapper.CORE_DIR + "/cfg/";
22                 return Directory.GetFiles(dir)
23                     .Where(s => s.EndsWith(".cfg"))
24                     .Select(s => s.Replace(dir, "").Replace(".cfg", ""))
25                     .ToArray();
26             }
27         }
28
29         public static Dictionary<string, Type> Modules = new Dictionary<string, Type>() {
30             { "Analyze", typeof(CASP_AnalyzeForm) },
31             //{ "Lint", null },
32             { "Outline", typeof(CASP_OutlineForm) },
33             { "Print", typeof(CASP_PrintForm) },
34             { "Translate", typeof(CASP_TranslateForm) }
35         };
36
37         public static string TempFilename = "CASP_Temp_Src.tmp";
38
39         public string request;
40         private List<KeyValuePair<string, string>> customArgs = new List<KeyValuePair<string, string>>();
41
42
43         private void UpdateRequest()
44         {
45             try
46             {
47                 string module = ConsoleWrapper.GetArgument(ConsoleWrapper.ModuleId, ModuleCombo.SelectedItem.ToString());
48                 string srclang = ConsoleWrapper.GetArgument(ConsoleWrapper.SourceLanguage, InputLanguageCombo.SelectedItem.ToString());
49                 string code = ConsoleWrapper.GetArgument(ConsoleWrapper.CodeFile, TempFilename);
50                 //string code = ConsoleWrapper.GetArgument(ConsoleWrapper.CodeSnippet, InputTextbox.Text);
51                 List<string> requestData = customArgs.Select(kvp => ConsoleWrapper.GetArgument(kvp.Key, kvp.Value)).ToList();
52                 requestData.Insert(0, code);
53                 requestData.Insert(0, srclang);
54                 requestData.Insert(0, module);
55
56                 //request = ConsoleWrapper.GenerateRequest(module, srclang, code);
57                 request = ConsoleWrapper.GenerateRequest(requestData.ToArray());
58                 RequestTextbox.Text = "CASP " + request;
59
60             }
61             catch (Exception e)
62             {
63
64             }
65         }
66
```

```
67          private async void Execute()
68          {
69              if (!ConsoleWrapper.Running)
70              {
71                  string filename = ConsoleWrapper.CORE_DIR + "/" + TempFilename;
72                  File.WriteAllText(filename, InputTextbox.Text);
73
74                  SetExecute(true);
75                  ProgramStatus.Text = "Processing...";
76                  string output = await ConsoleWrapper.Execute(request);
77                  ProgramStatus.Text = "Ready (" + ((float)ConsoleWrapper.LastRunTime / 1000f) + "s)";
78                  SetExecute(false);
79
80                  if (ShowOutputCheckbox.Checked)
81                      new OutputForm(output).Show();
82
83                  Type T = Modules[ModuleCombo.SelectedItem.ToString()];
84                  if (T != null && T.IsSubclassOf(typeof(CASP_OutputForm)))
85                  {
86                      Regex reg = new Regex("CASP_RETURN_DATA_START(.*)CASP_RETURN_DATA_END", RegexOptions.Singleline);
87                      string jsonString = reg.Match(output).Groups[1].Value.Trim();
88                      JObject response = JsonConvert.DeserializeObject<JObject>(jsonString);
89
90                      if (response != null)
91                      {
92                          CASP_OutputForm form = (CASP_OutputForm)Activator.CreateInstance(T);
93                          form.Show();
94                          form.Set_CASP_Output(response);
95                      }
96                      else
97                      {
98                          response = JsonConvert.DeserializeObject<JObject>("{ \"Data\": {}, \"Warnings\": [], \"Errors\": [ { \"id\": -1, \"message\": \"CASP produced no valid output.\" }
99                      }
100
101                     ErrorProviderForm errorProvider = new ErrorProviderForm(response);
102                     if (errorProvider.NumErrors > 0 || errorProvider.NumWarnings > 0)
103                         errorProvider.Show();
104                     else
105                         errorProvider.Dispose();
106                 }
107
108                 File.Delete(filename);
109             }
110             else
111             {
112                 ConsoleWrapper.Kill();
113                 SetExecute(false);
114             }
115         }
116
117         public MainForm()
118         {
119             InitializeComponent();
120         }
121
122         private void MainForm_Load(object sender, EventArgs e)
123         {
124             ProgramStatus.Text = "Ready";
125
126             ModuleCombo.Items.AddRange(Modules.Select(entry => entry.Key).ToArray());
127             ModuleCombo.SelectedItem = ModuleCombo.Items[0];
128
129             InputLanguageCombo.Items.AddRange(Languages);
130             InputLanguageCombo.SelectedItem = InputLanguageCombo.Items[0];
131
132             UpdateRequest();
133         }
```

```
134
135         private void SelectedIndexChanged(object sender, EventArgs e)
136         {
137             UpdateRequest();
138         }
139
140         private void SetExecute(bool active)
141         {
142             ExecuteButton.Text = active ? "Stop" : "Execute Command";
143         }
144
145         private void ExecuteButton_Click(object sender, EventArgs e)
146         {
147             Execute();
148         }
149
150         private void InputTextbox_TextChanged(object sender, EventArgs e)
151         {
152
153         }
154
155         private void NewArgButton_Click(object sender, EventArgs e)
156         {
157             AddArgForm newArg = new AddArgForm(AddArgument);
158             newArg.Show();
159         }
160
161         private void AddArgument(string argName, string argValue)
162         {
163             customArgs.Add(new KeyValuePair<string, string>(argName, argValue));
164             UpdateArguments();
165         }
166
167         private void UpdateArguments()
168         {
169             OtherArgs.Items.Clear();
170             foreach (KeyValuePair<string, string> kvp in customArgs)
171             {
172                 OtherArgs.Items.Add(kvp.Key + ": " + kvp.Value);
173             }
174             UpdateRequest();
175         }
176
177         private void RemoveArgs_Click(object sender, EventArgs e)
178         {
179             ListBox.SelectedIndexCollection indices = OtherArgs.SelectedIndices;
180             for (int i = indices.Count - 1; i >= 0; i--)
181             {
182                 customArgs.RemoveAt(indices[i]);
183             }
184             UpdateArguments();
185         }
186     }
187 }
```

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.ComponentModel;
 4 using System.Data;
 5 using System.Drawing;
 6 using System.Linq;
 7 using System.Text;
 8 using System.Threading.Tasks;
 9 using System.Windows.Forms;
10
11 namespace CASP_Standalone_Implementation.Forms
12 {
13     public partial class OutputForm : Form
14     {
15         public OutputForm(string text)
16         {
17             InitializeComponent();
18
19             OutputTextbox.Text = text;
20         }
21     }
22 }
```

```
 1 using Newtonsoft.Json.Linq;
 2 using System.Windows.Forms;
 3
 4 namespace CASP_Standalone_Implementation.Src
 5 {
 6     public abstract class CASP_OutputForm : Form
 7     {
 8         public abstract void Set_CASP_Output(JObject CASP_Response);
 9     }
10 }
```

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Diagnostics;
 4 using System.IO;
 5 using System.Linq;
 6 using System.Text;
 7 using System.Threading.Tasks;
 8
 9 namespace CASP_Standalone_Implementation.Src
10 {
11     class ConsoleWrapper
12     {
13         public static string CORE_DIR = Path.GetFullPath("../../../../../Core");
14
15         public static string ModuleId = "moduleid";
16         public static string CodeSnippet = "code";
17         public static string CodeFile = "codef";
18         public static string FunctionArgument = "args";
19         public static string SourceLanguage = "sourcelang";
20
21         public static bool Running
22         {
23             get
24             {
25                 return ActiveProcess != null;
26             }
27         }
28
29         static Process ActiveProcess = null;
30         static long lastRunTime = 0;
31
32         public static long LastRunTime
33         {
34             get
35             {
36                 return lastRunTime;
37             }
38         }
39
40         public static string GenerateRequest(params string[] data)
41         {
42             string ret = "";
43
44             foreach (string arg in data)
45             {
46                 ret += arg + " ";
47             }
48
49             return ret.Trim();
50         }
51
52         public static string GetArgument(string type, string data)
53         {
54             return "/" + type + "=\"" + data + "\"";
55         }
56
57         public static async Task<string> Execute(string request)
58         {
59             Stopwatch timer = new Stopwatch();
60             return await Task.Run(() =>
61             {
62                 string output = "";
63                 Kill();
64                 try {
65                     ActiveProcess = new Process();
66                     ActiveProcess.StartInfo.UseShellExecute = false;
```

```
67                      ActiveProcess.StartInfo.RedirectStandardOutput = true;
68                      ActiveProcess.StartInfo.CreateNoWindow = true;
69                      ActiveProcess.StartInfo.WorkingDirectory = CORE_DIR;
70                      ActiveProcess.StartInfo.FileName = CORE_DIR + "/CASP.exe";
71                      ActiveProcess.StartInfo.Arguments = request;
72
73                      timer.Start();
74
75                      ActiveProcess.Start();
76                      output = ActiveProcess.StandardOutput.ReadToEnd();
77                      ActiveProcess.WaitForExit();
78                  }
79                  catch (Exception e)
80                  {
81                      // If a process is forcefully killed in the middle of operation, it might throw an error
82                      ActiveProcess.Dispose();
83                      ActiveProcess = null;
84                  }
85
86                  timer.Stop();
87                  lastRunTime = timer.ElapsedMilliseconds;
88
89                  Kill();
90
91                  return output;
92              });
93          }
94
95          public static void Kill()
96          {
97              if (Running)
98              {
99                  if (!ActiveProcess.HasExited)
100                     ActiveProcess.Kill();
101                 ActiveProcess.Dispose();
102                 ActiveProcess = null;
103             }
104         }
105     }
106 }
```

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Drawing;
 4 using System.Drawing.Drawing2D;
 5 using System.Linq;
 6 using System.Text;
 7 using System.Threading.Tasks;
 8 using System.Windows.Forms;
 9
10 namespace CASP_Standalone_Implementation.Src
11 {
12     public enum BlockType { Start, MethodCall, Process, Loop, Decision, EndDecision, IO, End };
13
14     public class OutlineGraph
15     {
16         public List<OutlineNode> nodes = new List<OutlineNode>();
17         public List<OutlineEdge> edges = new List<OutlineEdge>();
18
19         public OutlineNode AddNode(OutlineNode node)
20         {
21             node.index = nodes.Count;
22             nodes.Add(node);
23             return node;
24         }
25
26         public OutlineEdge AddEdge(int sourceIndex, int targetIndex, string text = "")
27         {
28             OutlineEdge edge = nodes[sourceIndex].AddEdge(nodes[targetIndex], text);
29             edges.Add(edge);
30             return edge;
31         }
32
33         public void Reset()
34         {
35             for (int i = 0; i < nodes.Count; i++)
36             {
37                 nodes[i].drawn = false;
38             }
39         }
40     }
41
42     public class OutlineNode
43     {
44         public bool drawn = false;
45         public int index;
46         public string text;
47         public BlockType type;
48         public List<OutlineEdge> edges = new List<OutlineEdge>();
49
50         public OutlineEdge AddEdge(OutlineNode target, string text = "")
51         {
52             OutlineEdge edge = new OutlineEdge() { source = this, target = target, text = text };
53             edges.Add(edge);
54             return edge;
55         }
56     }
57
58     public class OutlineEdge
59     {
60         public string text;
61         public OutlineNode source;
62         public OutlineNode target;
63     }
64
65     public class FlowBlock : Label
66     {
```

```
67          Socket TopSocket;
68          Socket BottomSocket;
69          Socket LeftSocket;
70          Socket RightSocket;
71
72          public bool MouseOver = false;
73          bool renderingSockets = false;
74          public BlockType type;
75          public FlowBlock parent = null;
76          public List<FlowBlock> children = new List<FlowBlock>();
77
78          public List<FlowBlock> siblings
79          {
80              get
81              {
82                  List<FlowBlock> sibs = new List<FlowBlock>();
83                  if (parent != null)
84                  {
85                      for (int i = 0; i < parent.children.Count; i++)
86                      {
87                          if (parent.children[i] != this)
88                              sibs.Add(parent.children[i]);
89                      }
90                  }
91                  return sibs;
92              }
93          }
94
95          public int id;
96
97          public Point Center
98          {
99              get
100             {
101                 return new Point(Width / 2 + Location.X, Height / 2 + Location.Y);
102             }
103         }
104
105
106         public bool RenderSockets
107         {
108             get
109             {
110                 return renderingSockets;
111             }
112             set
113             {
114                 if (value != renderingSockets)
115                 {
116                     if (value)
117                         Paint += FlowBlock_Paint;
118                     else
119                         Paint -= FlowBlock_Paint;
120                 }
121                 renderingSockets = value;
122             }
123         }
124
125         public FlowBlock() : base() {
126             TopSocket = new Socket(this, 90);
127             BottomSocket = new Socket(this, 270);
128             LeftSocket = new Socket(this, 180);
129             RightSocket = new Socket(this, 0);
130             UpdateSockets();
131             //RenderSockets = true;
132         }
133
```

```
134         public void UpdateSockets()
135         {
136             int halfWidth = Width / 2;
137             int halfHeight = Height / 2;
138             int Top = 0;
139             int Bottom = Height;
140             int Left = 0;
141             int Right = Width;
142 
143             TopSocket.SetLocation(halfWidth, Top);
144             BottomSocket.SetLocation(halfWidth, Bottom);
145             LeftSocket.SetLocation(Left, halfHeight);
146             RightSocket.SetLocation(Right, halfHeight);
147         }
148 
149         public Socket ClosestSocketToPoint(Point pt)
150         {
151             Socket closest = TopSocket;
152             double min = TopSocket.DistanceToPoint(pt);
153             double next = BottomSocket.DistanceToPoint(pt);
154             if (next < min)
155             {
156                 min = next;
157                 closest = BottomSocket;
158             }
159             next = LeftSocket.DistanceToPoint(pt);
160             if (next < min)
161             {
162                 min = next;
163                 closest = LeftSocket;
164             }
165             next = RightSocket.DistanceToPoint(pt);
166             if (next < min)
167             {
168                 min = next;
169                 closest = RightSocket;
170             }
171             return closest;
172         }
173 
174         public Socket ClosestUnusedSocketToPoint(Point pt)
175         {
176             Socket closest = null;
177             double min = double.PositiveInfinity;
178             double next = 0;
179 
180             if (TopSocket.Unused)
181             {
182                 next = TopSocket.DistanceToPoint(pt);
183                 if (next < min)
184                 {
185                     min = next;
186                     closest = TopSocket;
187                 }
188             }
189             if (BottomSocket.Unused)
190             {
191                 next = BottomSocket.DistanceToPoint(pt);
192                 if (next < min)
193                 {
194                     min = next;
195                     closest = BottomSocket;
196                 }
197             }
198             if (LeftSocket.Unused) {
199                 next = LeftSocket.DistanceToPoint(pt);
200                 if (next < min)
```

```
201                 {
202                     min = next;
203                     closest = LeftSocket;
204                 }
205             }
206             if (RightSocket.Unused) {
207                 next = RightSocket.DistanceToPoint(pt);
208                 if (next < min)
209                 {
210                     min = next;
211                     closest = RightSocket;
212                 }
213             }
214             return closest;
215         }
216 
217         void ClosestSocketPair(FlowBlock block, out Socket sourceSocket, out Socket targetSocket)
218         {
219             Socket source = null;
220             Socket target = null;
221 
222             double min = double.PositiveInfinity;
223             double next;
224 
225             if (TopSocket.Unused)
226             {
227                 Socket temp = block.type != BlockType.EndDecision ?
228                     block.ClosestUnusedSocketToPoint(TopSocket.Location) :
229                     block.ClosestSocketToPoint(TopSocket.Location);
230                 if (temp != null)
231                 {
232                     next = TopSocket.DistanceToPoint(temp.Location);
233                     if (next < min)
234                     {
235                         min = next;
236                         source = TopSocket;
237                         target = temp;
238                     }
239                 }
240             }
241 
242             if (BottomSocket.Unused)
243             {
244                 Socket temp = block.type != BlockType.EndDecision ?
245                     block.ClosestUnusedSocketToPoint(BottomSocket.Location) :
246                     block.ClosestSocketToPoint(BottomSocket.Location);
247                 if (temp != null)
248                 {
249                     next = BottomSocket.DistanceToPoint(temp.Location);
250                     if (next < min)
251                     {
252                         min = next;
253                         source = BottomSocket;
254                         target = temp;
255                     }
256                 }
257             }
258 
259             if (LeftSocket.Unused)
260             {
261                 Socket temp = block.type != BlockType.EndDecision ?
262                     block.ClosestUnusedSocketToPoint(LeftSocket.Location) :
263                     block.ClosestSocketToPoint(LeftSocket.Location);
264                 if (temp != null)
265                 {
266                     next = LeftSocket.DistanceToPoint(temp.Location);
267                     if (next < min)
```

```
268                     {
269                         min = next;
270                         source = LeftSocket;
271                         target = temp;
272                     }
273                 }
274             }
275 
276             if (RightSocket.Unused)
277             {
278                 Socket temp = block.type != BlockType.EndDecision ?
279                     block.ClosestUnusedSocketToPoint(RightSocket.Location) :
280                     block.ClosestSocketToPoint(RightSocket.Location);
281                 if (temp != null)
282                 {
283                     next = RightSocket.DistanceToPoint(temp.Location);
284                     if (next < min)
285                     {
286                         min = next;
287                         source = RightSocket;
288                         target = temp;
289                     }
290                 }
291             }
292 
293             sourceSocket = source;
294             targetSocket = target;
295         }
296 
297         public bool ConnectTo(FlowBlock block, string data = "")
298         {
299             Socket sourceSocket;
300             Socket targetSocket;
301             ClosestSocketPair(block, out sourceSocket, out targetSocket);
302 
303             if (sourceSocket != null && targetSocket != null)
304             {
305                 Connector c = new Connector()
306                 {
307                     data = data
308                 };
309                 sourceSocket.ConnectAsSource(c);
310                 targetSocket.ConnectAsTarget(c);
311                 return true;
312             }
313             return false;
314         }
315 
316         private void FlowBlock_Paint(object sender, PaintEventArgs e)
317         {
318             Pen pen = Pens.Black;
319             Brush brush = Brushes.Black;
320 
321             int size = 10;
322             int hs = size / 2;
323 
324             if (TopSocket.Unused)
325                 e.Graphics.DrawEllipse(pen, new Rectangle(TopSocket.LocalLocation.X - hs, TopSocket.LocalLocation.Y - hs, size, size));
326             else
327                 e.Graphics.FillEllipse(brush, new Rectangle(TopSocket.LocalLocation.X - hs, TopSocket.LocalLocation.Y - hs, size, size));
328 
329             if (BottomSocket.Unused)
330                 e.Graphics.DrawEllipse(pen, new Rectangle(BottomSocket.LocalLocation.X - hs, BottomSocket.LocalLocation.Y - hs, size, size));
331             else
332                 e.Graphics.FillEllipse(brush, new Rectangle(BottomSocket.LocalLocation.X - hs, BottomSocket.LocalLocation.Y - hs, size, size));
333 
334             if (LeftSocket.Unused)
```

```
335                 e.Graphics.DrawEllipse(pen, new Rectangle(LeftSocket.LocalLocation.X - hs, LeftSocket.LocalLocation.Y - hs, size, size));
336             else
337                 e.Graphics.FillEllipse(brush, new Rectangle(LeftSocket.LocalLocation.X - hs, LeftSocket.LocalLocation.Y - hs, size, size));
338 
339             if (RightSocket.Unused)
340                 e.Graphics.DrawEllipse(pen, new Rectangle(RightSocket.LocalLocation.X - hs, RightSocket.LocalLocation.Y - hs, size, size));
341             else
342                 e.Graphics.FillEllipse(brush, new Rectangle(RightSocket.LocalLocation.X - hs, RightSocket.LocalLocation.Y - hs, size, size));
343 
344         }
345 
346         public void RenderEdgeGraphics(Graphics g)
347         {
348             if (!TopSocket.Unused && TopSocket.isSource)
349                 TopSocket.Connector.RenderGraphicsPath(g);
350 
351             if (!BottomSocket.Unused && BottomSocket.isSource)
352                 BottomSocket.Connector.RenderGraphicsPath(g);
353 
354             if (!LeftSocket.Unused && LeftSocket.isSource)
355                 LeftSocket.Connector.RenderGraphicsPath(g);
356 
357             if (!RightSocket.Unused && RightSocket.isSource)
358                 RightSocket.Connector.RenderGraphicsPath(g);
359         }
360     }
361 
362     public class Socket
363     {
364         public FlowBlock FlowBlock;
365         public Point LocalLocation;
366         public Connector Connector = null;
367         public double angle = 0;
368         public double outAngle
369         {
370             get
371             {
372                 return angle % 360;
373             }
374         }
375         public double inAngle
376         {
377             get
378             {
379                 return (outAngle + 180) % 360;
380             }
381         }
382 
383         public bool isSource
384         {
385             get
386             {
387                 if (Connector != null)
388                     return Connector.Source == this;
389                 return false;
390             }
391         }
392 
393         public bool isTarget
394         {
395             get
396             {
397                 if (Connector != null)
398                     return Connector.Target == this;
399                 return false;
400             }
401         }
```

```
402
403         public Point Location
404         {
405             get
406             {
407                 Point pt = new Point(0, 0);
408                 pt.Offset(FlowBlock.Location);
409                 pt.Offset(LocalLocation);
410                 return pt;
411             }
412         }
413
414         public bool Unused
415         {
416             get
417             {
418                 return Connector == null;
419             }
420         }
421
422         public Socket(FlowBlock parent, double angle, int localX = 0, int localY = 0)
423         {
424             FlowBlock = parent;
425             LocalLocation = new Point(localX, localY);
426             this.angle = angle;
427         }
428
429         public void SetLocation(int localX, int localY)
430         {
431             LocalLocation = new Point(localX, localY);
432         }
433
434         public double DistanceToPoint(Point pt)
435         {
436             return Math.Sqrt(Math.Pow(pt.X - Location.X, 2) + Math.Pow(pt.Y - Location.Y, 2));
437         }
438
439         public bool ConnectAsSource(Connector connector)
440         {
441             //if (Connector == null)
442             //{
443             //    if (connector.Source != null)
444             //        connector.Source.Connector = null;
445                 connector.Source = this;
446                 Connector = connector;
447                 return true;
448             //}
449             //return false;
450         }
451
452         public bool ConnectAsTarget(Connector connector)
453         {
454             //if (Connector == null)
455             //{
456             //    if (connector.Target != null)
457             //        connector.Target.Connector = null;
458                 connector.Target = this;
459                 Connector = connector;
460                 return true;
461             //}
462             //return false;
463         }
464
465     }
466
467     public class Connector
468     {
```

```
469         public Socket Source;
470         public Socket Target;
471 
472         public string data;
473 
474         public void RenderGraphicsPath(Graphics g)
475         {
476             if (Source != null && Target != null)
477             {
478                 RenderArrow(g, Pens.SlateGray, Brushes.SlateGray, 60, 8);
479 
480                 int fontSize = 10;
481                 PointF pt = Source.Location;
482                 StringFormatFlags flags = StringFormatFlags.NoWrap;
483                 if (Source.outAngle == 0 || Source.outAngle == 180)
484                 {
485                     flags = flags | StringFormatFlags.DirectionVertical;
486                 }
487                 StringFormat sf = new StringFormat(flags);
488                 FontFamily fam = new FontFamily("microsoft sans serif");
489                 Font font = new Font(fam, fontSize);
490 
491                 SizeF size = g.MeasureString(data, font);
492 
493                 if (Source.outAngle == 0)
494                     pt = new PointF(pt.X + 3 - size.Height / 2, pt.Y - size.Width / 2);
495                 else if (Source.outAngle == 180)
496                     pt = new PointF(pt.X - 3 - size.Height / 2, pt.Y - size.Width / 2);
497                 else if (Source.outAngle == 90)
498                     pt = new PointF(pt.X - size.Width / 2, pt.Y - 3 - size.Height / 2);
499                 else if (Source.outAngle == 270)
500                     pt = new PointF(pt.X - size.Width / 2, pt.Y + 3 - size.Height / 2);
501 
502 
503                 g.DrawString(data, font, Brushes.Black, pt, sf);
504             }
505         }
506 
507         private void RenderArrow(Graphics g, Pen pen, Brush brush, int arrowAngle, int arrowLength)
508         {
509             Point end = Source.Location;
510 
511             double outAngle = Source.outAngle;
512             double inAngle = Target.inAngle;
513 
514             // complimentary sides (t & b, l & r)
515             if (outAngle == inAngle)
516             {
517                 // vertical
518                 if (Math.Abs(outAngle % 180) == 90)
519                 {
520                     if (Source.Location.X != Target.Location.X)
521                     {
522                         int hy = Source.Location.Y + (Target.Location.Y - Source.Location.Y) / 2;
523                         Point p1 = Source.Location;
524                         Point p2 = new Point(Source.Location.X, hy);
525                         Point p3 = new Point(Target.Location.X, hy);
526                         g.DrawLine(pen, p1, p2);
527                         g.DrawLine(pen, p2, p3);
528                         end = p3;
529                     }
530                 }
531                 // horizontal
532                 else if (Math.Abs(outAngle % 180) == 0)
533                 {
534                     if (Source.Location.Y != Target.Location.Y)
535                     {
```

```
536                     int hx = Source.Location.X + (Target.Location.X - Source.Location.X) / 2;
537                     Point p1 = Source.Location;
538                     Point p2 = new Point(hx, Source.Location.Y);
539                     Point p3 = new Point(hx, Target.Location.Y);
540                     g.DrawLine(pen, p1, p2);
541                     g.DrawLine(pen, p2, p3);
542                     end = p3;
543                 }
544             }
545         }
546         // opposite sides (l & l, t & t, etc.)
547         else if ((inAngle + 180) % 360 == outAngle)
548         {
549             bool set = false;
550             Point p1 = Source.Location, p2 = new Point(), p3 = new Point();
551             // right
552             if (outAngle == 0)
553             {
554                 set = true;
555                 int maxX = Math.Max(Source.Location.X, Target.Location.X) + 10;
556                 p2 = new Point(maxX, Source.Location.Y);
557                 p3 = new Point(maxX, Target.Location.Y);
558             }
559             // left
560             else if (outAngle == 180)
561             {
562                 set = true;
563                 int minX = Math.Min(Source.Location.X, Target.Location.X) - 10;
564                 p2 = new Point(minX, Source.Location.Y);
565                 p3 = new Point(minX, Target.Location.Y);
566             }
567             // top
568             if (outAngle == 90)
569             {
570                 set = true;
571                 int minY = Math.Max(Source.Location.Y, Target.Location.Y) - 10;
572                 p2 = new Point(Source.Location.X, minY);
573                 p3 = new Point(Target.Location.X, minY);
574             }
575             // bottom
576             else if (outAngle == 270)
577             {
578                 set = true;
579                 int maxY = Math.Min(Source.Location.Y, Target.Location.Y) + 10;
580                 p2 = new Point(Source.Location.X, maxY);
581                 p3 = new Point(Target.Location.X, maxY);
582             }
583 
584             if (set)
585             {
586                 g.DrawLine(pen, p1, p2);
587                 g.DrawLine(pen, p2, p3);
588                 end = p3;
589             }
590         }
591         // different axes
592         else
593         {
594             Point p1 = Source.Location, p2 = new Point(), p3 = new Point(), p4 = new Point();
595             // out right or left
596             if (outAngle == 0 || outAngle == 180)
597             {
598                 // in bottom or top
599                 if (inAngle == 90 || inAngle == 270)
600                 {
601                     bool comp;
602                     int xOff = 10;
```

```
603                     if (outAngle == 0)
604                     {
605                         comp = Source.Location.X < Target.Location.X;
606                     }
607                     else
608                     {
609                         comp = Source.Location.X > Target.Location.X;
610                         xOff *= -1;
611                     }
612 
613                     if (comp)
614                     {
615                         p2 = new Point(Target.Location.X, Source.Location.Y);
616                         g.DrawLine(pen, p1, p2);
617                         end = p2;
618                     }
619                     else
620                     {
621                         int yOff = -10;
622                         if (inAngle == 90)
623                             yOff *= -1;
624 
625                         p2 = new Point(Source.Location.X + xOff, p1.Y);
626                         p3 = new Point(p2.X, Target.Location.Y + yOff);
627                         p4 = new Point(Target.Location.X, p3.Y);
628                         g.DrawLine(pen, p1, p2);
629                         g.DrawLine(pen, p2, p3);
630                         g.DrawLine(pen, p3, p4);
631                         end = p4;
632                     }
633                 }
634             }
635             // out bottom or top
636             if (outAngle == 90 || outAngle == 270)
637             {
638                 // in left or right
639                 if (inAngle == 0 || inAngle == 180)
640                 {
641                     bool comp;
642                     int yOff = 10;
643                     if (outAngle == 90)
644                     {
645                         comp = Source.Location.Y > Target.Location.Y;
646                         yOff *= -1;
647                     }
648                     else
649                     {
650                         comp = Source.Location.Y < Target.Location.Y;
651                     }
652 
653                     if (comp)
654                     {
655                         p2 = new Point(Source.Location.X, Target.Location.Y);
656                         g.DrawLine(pen, p1, p2);
657                         end = p2;
658                     }
659                     else
660                     {
661                         int xOff = 10;
662                         if (inAngle == 90)
663                             xOff *= -1;
664 
665                         p2 = new Point(p1.X, Source.Location.Y + yOff);
666                         p3 = new Point(Target.Location.X + xOff, p2.Y);
667                         p4 = new Point(p3.X, Target.Location.Y);
668                         g.DrawLine(pen, p1, p2);
669                         g.DrawLine(pen, p2, p3);
```

```
670                             g.DrawLine(pen, p3, p4);
671                             end = p4;
672                         }
673                     }
674                 }
675             }
676 
677             Point source = end;
678             Point target = Target.Location;
679 
680             double initialAngleDeg = Math.Atan2(target.Y - source.Y, target.X - source.X) * 180 / Math.PI;
681             double angleLRad = (initialAngleDeg - 180 + arrowAngle / 2) * Math.PI / 180;
682             double angleRRad = (initialAngleDeg - 180 - arrowAngle / 2) * Math.PI / 180;
683 
684             Point arrowL = new Point(target.X + (int)(arrowLength * Math.Cos(angleLRad)), target.Y + (int)(arrowLength * Math.Sin(angleLRad)));
685             Point arrowR = new Point(target.X + (int)(arrowLength * Math.Cos(angleRRad)), target.Y + (int)(arrowLength * Math.Sin(angleRRad)));
686 
687             g.FillPolygon(brush, new Point[] { target, arrowL, arrowR });
688             g.DrawLine(pen, source, target);
689         }
690     }
691 }
```