```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 }
```