```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 }
```