```
/*
 *  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) {

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

    while (regex_search(t, matches, r)) {
        string terminalID = matches[1].str();
        ignore[terminalID] = true;

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

void LanguageDescriptorObject::ParseFSM(string data) {

    string t = string(data);
    regex r = regex("^\\([ \t]*([a-zA-Z_0-9]+)[ \t]*,[ \t]*([^\t\n]+)[ \t]*\\)[ \t]*->[ \t]*([^ \t\n]+)$");
    smatch matches;

    while (regex_search(t, matches, r)) {
        string fromState = matches[1].str();
        string toState = matches[3].str();
        string chars = matches[2].str();

        int index = -1;
        while ((index = chars.find("\\", index + 1)) != -1) {
            if (index < chars.size() - 1) {
                chars = chars.substr(0, index) + chars.substr(index + 1, chars.size());
                switch (chars[index]) {
                    case 'n':
                        chars[index] = '\n';
                        break;
                    case 't':
                        chars[index] = '\t';
                        break;
                    case 'r':
                        chars[index] = '\r';
                        break;
                    case '0':
                        chars[index] = '\0';
                        break;
                }
            } else
                chars = chars.substr(0, index);
        }

        vector<char> stateTransitions;
        for (int i = 0; i < chars.size(); i++) {
            stateTransitions.push_back(chars[i]);
        }

        stateMachine.AddState(fromState);
        stateMachine.AddState(toState);
        stateMachine.AddTransition(fromState, toState, stateTransitions);

        // cout << "State " << matches[1] << " moves to state " << matches[3] << " with any of the following input: " << matches[2] << endl;
        t = matches.suffix().str();
    }

    t = string(data);
    r = regex("^F\\([ \t]*([^ \t\n]+)[ \t]*,[ \t]*([^ \t\n]+)[ \t]*\\)[ \t]*$");

    while (regex_search(t, matches, r)) {
        string target = matches[1].str();
        string token = matches[2].str();

        stateMachine.AddGoal(target, token);
        // cout << "State " << matches[1] << " accepts token " << matches[2] << endl;
        t = matches.suffix().str();
    }

    t = data;
    r = regex("^I\\([ \t]*([^ \t\n]+)[ \t]*\\)$");

    if (regex_search(t, matches, r)) {
        string target = matches[1].str();

        stateMachine.SetInitialState(target);

        // cout << "State " << matches[1] << " is the initial state" << endl;
        t = matches.suffix().str();
    }

    // stateMachine.Print();

}

vector<Token> LanguageDescriptorObject::Tokenize(string input) {
    vector<Token> tokens;
    string token;
    string tokenData;

    stateMachine.Reset();

    for (int i = 0; i < input.size(); i++) {
        tokenData += input[i];
        if ((token = stateMachine.Transition(input[i])) != "") {
            if (token == "ERROR") {
                if (input[i] != ' ' && input[i] != '\n' && input[i] != '\r' && input[i] != '\t')
                    cout << "State machine encountered an error on character '" << input[i] << "'\n";
            } else {
                tokenData.pop_back();

                if (reservedWords[tokenData] != "")
                    token = reservedWords[tokenData];

                if (!IsTerminalIgnored(token))
                    tokens.push_back(Token(token, tokenData));
                else
                    cout << "Ignoring terminal " << token << ", value = \"" << tokenData << "\"" << endl;
                i--;
            }
            tokenData = "";
        }
    }

    if (token == "") {
        // accept the last token, only if there is one to accept
        token = stateMachine.Transition('\0');
        if (token == "" || token == "ERROR") {
            cout << "State machine encountered an error on character 'EOF'\n";
        } else {
            if (reservedWords[tokenData] != "")
                token = reservedWords[tokenData];

            if (!IsTerminalIgnored(token))
                tokens.push_back(Token(token, tokenData));
            else
                cout << "Ignoring terminal " << token << ", value = \"" << tokenData << "\"" << endl;
        }
    }
    stateMachine.Reset();

    // for (int i = 0; i < tokens.size(); i++) {
    //     tokens[i].Print();
    // }

    return tokens;

}

vector<Token> LanguageDescriptorObject::Tokenize(Markup* input) {
    vector<Token> tokens;

    if (!input->IsLeaf()) {
        vector<Markup*> children = input->Children();

        for (int i = 0; i < children.size(); i++) {
            vector<Token> tl = Tokenize(children[i]);
            tokens.insert(tokens.end(), tl.begin(), tl.end());
        }
    } else {
        Token t(input->GetID(), input->GetData());
        tokens.push_back(t);
    }

    return tokens;

}

LanguageDescriptorObject::LanguageDescriptorObject()
{

}

LanguageDescriptorObject::LanguageDescriptorObject(string language)
{
    Parse(language);
}

LanguageDescriptorObject::~LanguageDescriptorObject() {

}

void LanguageDescriptorObject::Parse(string language) {
    // getpath function ?
    string file = CFG_DIR + language + CFG_EXT;
    FILE* temp = fopen((file).c_str(), "r");
    if (temp != NULL) {
        fclose(temp);
    } else {
        // try PATH environment variable?
        throw "Cannot find language file.";
    }
    // return file;
    //
    this->language = language;

    string data = Helpers::ReadFile(file); // TODO: file data should probably already be passed in?
    string t = data;
    ParseTerminalValues(data);
    ParseFSM(data);
    ParseReservedWords(data);
    ParseIgnores(data);

    regex r = regex("(.+?)\\s*=:\\s*([^]+?)\\n\\n");
    smatch matches;

    while (regex_search(t, matches, r)) {
        Production* prod = new Production(this, matches[1], matches[2]);
        productions.push_back(prod);
        t = matches.suffix().str();
    }

    // for (int i = 0; i < productions.size(); i++) {
    //     cout << productions[i]->GetId() << ": " << productions[i]->GetRegex() << endl << endl;
    // }

}

vector<Production*> LanguageDescriptorObject::GetProductions() {
    return productions;
}

Production* LanguageDescriptorObject::findProdById(string id) {
    for (int i = 0; i < productions.size(); i++) {
        if (productions[i]->GetId() == id) {
            return productions[i];
        }
    }
    return NULL;
}
int LanguageDescriptorObject::getProdIndex(string id) {
    for (int i = 0; i < productions.size(); i++) {
        if (productions[i]->GetId() == id) {
            return i;
        }
    }
    return -1;
}

string LanguageDescriptorObject::GetLanguage() {
    return language;
}

vector<Production*> LanguageDescriptorObject::GetOrderedProductions(vector<string> stringlist) {
    vector<Production*> v;

    int size = stringlist.size();
    int* indexer = (int*)calloc(size, sizeof(int));
    int i;
    for (i = 0; i < size; i++) {
        indexer[i] = getProdIndex(stringlist[i]);
    }

    for (i = 1; i < size; i++) {
        if (i > 0 && indexer[i - 1] < indexer[i]) {
            string temps = stringlist[i];
            stringlist[i] = stringlist[i - 1];
            stringlist[i - 1] = temps;
            int tempi = indexer[i];
            indexer[i] =indexer[i - 1];
            indexer[i - 1] = tempi;
            i-=2;
        }
    }

    for (i = 0; i < size; i++) {
        if (i == 0 || indexer[i] != indexer[i-1]) {
            v.push_back(findProdById(stringlist[i]));
        }
    }

    return v;
}
```

```
Production::Production(LanguageDescriptorObject* ob, string id, string data) {
    ldo = ob;
    Parse(id, data);
}

TokenMatch* Production::Match(vector<Token> tokens) {
    return Match(tokens, 0);
}

TokenMatch* Production::Match(vector<Token> tokens, int start) {
    TokenMatch* t = rootSet->Match(tokens, start);
    return t;
}

TokenMatch* Production::MatchStrict(vector<Token> tokens) {
    return MatchStrict(tokens, 0);
}

TokenMatch* Production::MatchStrict(vector<Token> tokens, int start) {
    TokenMatch* t = rootSet->MatchStrict(tokens, start);
    return t;
}

void Production::Parse(string id, string data) {
    this->id = id;
    this->data = data;

    rootSet = new ProductionSet(this);
    rootSet->Parse(data);
}

string Production::GetRegex() {

    vector<Production*> prods = GetContainedProductions();
    string t = data;
    regex r;
    smatch matches;

    for (int i = 0; i < prods.size(); i++) {
        Production* prod = prods[i];
        string sub = "(?:" + prod->GetRegex() + ")";
        r = regex("<" + prod->GetId() + ">");

        while (regex_search(t, matches, r)) {
            t = matches.prefix().str() + sub + matches.suffix().str();
        }
    }

    return t;
}

string Production::GetId() {
    return id;
}

vector<Production*> Production::GetContainedProductions() {
    vector<Production*> prods;
    for (int i = 0; i < subproductions.size(); i++) {
        Production* p = ldo->findProdById(subproductions[i]);
        if (p != NULL) {
            prods.push_back(p);
        }
    }
    return prods;
}

LanguageDescriptorObject* Production::GetLDO() {
    return ldo;
}

ProductionSet* Production::GetRootProductionSet() {
    return rootSet;
}

ProductionSet::ProductionSet(Production* parentProduction) {
    prod = parentProduction;
}

void ProductionSet::Parse(string data) {
    source = data;

    string a = "(?:\\$([^\\$]*?)\\$)"; // Action Routine
    // string g = "(?:\\(([^\\)]*?)\\))"; // Group
    string te = "(?:\\[(.*?)\\])"; // Terminal
    string p = "(?:<(.*?)>)"; // Production
    string m = "(\\?|\\*|\\+)"; // Multiplicity
    string one = "(" + a + "|" + te + "|" + p + ")[\t ]*" + m + "?"; // One match
    string alt = "(?:[ \t]*\\|[ \t]*)"; // Alternation sequence
    string mult = "(?:" + alt + one + ")*"; // Multiple alternations
    string reg;
    if (type != _Alternation) {
        reg = "(" + one + ")(" + mult + ")";
    } else {
        reg = "(" + one + ")()";
    }

    regex r = regex(reg);
    smatch matches;
    string t = data;

    while (regex_search (t, matches, r)) {
        ProductionSet* newSet = new ProductionSet(prod);

        string actionRoutine = matches[3].str();
        string terminal = matches[4].str();
        string production = matches[5].str();
        string multiplicity = matches[6].str();
        string alternation = matches[7].str();

        if (alternation == "") {
            if (actionRoutine != "") {
                newSet->SetAction(actionRoutine);
            } else if (terminal != "") {
                newSet->SetTerminal(terminal);
            } else if (production != "") {
                newSet->SetProduction(production);
            }
            newSet->SetMultiplicity(multiplicity);
        } else {
            newSet->SetAlternation(matches[0]);
        }
        children.push_back(newSet);

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

}

void ProductionSet::SetAction(string data) {
    type = _Action;
    source = data;
}

void ProductionSet::SetTerminal(string data) {
```

```cpp
    type = _Terminal;
    source = data;
}

void ProductionSet::SetProduction(string data) {
    type = _Production;
    source = data;
}

void ProductionSet::SetAlternation(string data) {
    type = _Alternation;
    Parse(data);
}

void ProductionSet::SetMultiplicity(string data) {
    multiplicity = data;
}

TokenMatch* ProductionSet::Match(vector<Token> tokens) {
    return Match(tokens, 0);
}

TokenMatch* ProductionSet::Match(vector<Token> tokens, int startIndex) {
    TokenMatch* match;

    for (int tokenIndex = startIndex; tokenIndex < tokens.size(); tokenIndex++) {
        match = MatchStrict(tokens, tokenIndex);
        if (match != NULL) {
            return match;
        }
    }

    return NULL;
}

Production* ProductionSet::GetProduction() {
    return prod;
}

ProductionSetType ProductionSet::GetType() {
    return type;
}

vector<ProductionSet*> ProductionSet::GetChildren() {
    return children;
}

string ProductionSet::GetSource() {
    return source;
}

string ProductionSet::GetMultiplicity() {
    return multiplicity;
}



TokenMatch* ProductionSet::MatchStrict(vector<Token> tokens, int startIndex) {
    TokenMatch* t = NULL;

    if (type == _Terminal) {
        t = MatchTerminal(tokens, startIndex);
    }
    else if (type == _Alternation) {
        t = MatchAlternation(tokens, startIndex);
    }
    else if (type == _Group || type == _Root) {
        t = MatchGroup(tokens, startIndex);
        if (type == _Root && t != NULL) {
            t->prod = GetProduction()->GetId();
    //     cout << "Matched " << t->prod << endl;
    //     cout << "Matched (" << source << "): count = " << t->length << ", start = " << t->begin << ", end = " << t->end << endl;
    // for (int p = 0; p < t->match.size(); p++) {
    //     cout << "\t" << t->match[p].id << endl;
    // }
    // cout << endl;
        }
    }
    else if (type == _Production) {
        t = MatchProduction(tokens, startIndex);
    } else if (type == _Action) {
        t = MatchAction(source, startIndex);
    }

    return t;
}
TokenMatch* ProductionSet::MatchAction(string source, int startIndex) {

    TokenMatch* match = new TokenMatch();

    match->begin = startIndex;
    match->end = startIndex;
    match->length = 0;
    match->isAction = true;
    match->prod = source;

    return match;
}

TokenMatch* ProductionSet::MatchGroup(vector<Token> tokens, int startIndex) {

    TokenMatch* match = new TokenMatch();
    bool isMatch = true, matched = true;
    int i = startIndex;

    TokenMatch* groupMatch;
    for (int j = 0; j < children.size(); j++) {
        groupMatch = children[j]->MatchStrict(tokens, i);
        if (groupMatch == NULL) {
            matched = false;
            match->submatches.clear();
            break;
        }
        if (groupMatch->length > 0 || groupMatch->isAction) {
            match->submatches.push_back(groupMatch);
            i += groupMatch->length;
        }
    }

    isMatch = multiplicity != "" || matched;

    if (!isMatch)
        return NULL;

    match->begin = startIndex;
    match->end = i;
    match->length = match->end - match->begin;
    match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);

    return match;
}
TokenMatch* ProductionSet::MatchTerminal(vector<Token> tokens, int startIndex) {

    if (startIndex >= tokens.size())
```

```
        return NULL;

    TokenMatch* match = new TokenMatch();
    bool isMatch = true, matched = false;

    matched = tokens[startIndex].id == source;
    isMatch = multiplicity != "" || matched;

    if (!isMatch)
        return NULL;

    match->begin = startIndex;
    match->end = startIndex + (matched ? 1 : 0);
    match->length = match->end - match->begin;
    match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);

    return match;
}
TokenMatch* ProductionSet::MatchAlternation(vector<Token> tokens, int startIndex) {

    TokenMatch* match = new TokenMatch();
    bool isMatch = true, matched = false;
    int i = startIndex;

    TokenMatch* alternationMatch = NULL;
    for (int j = 0; j < children.size(); j++) {
        alternationMatch = children[j]->MatchStrict(tokens, i);
        if (alternationMatch != NULL) {
            matched = true;
            if (alternationMatch->length > 0) {
                i += alternationMatch->length;
                match->submatches.push_back(alternationMatch);
                break;
            }
        }
    }

    isMatch = multiplicity != "" || matched;

    if (!isMatch)
        return NULL;

    match->begin = startIndex;
    match->end = i;
    match->length = match->end - match->begin;
    match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);

    return match;
}
TokenMatch* ProductionSet::MatchProduction(vector<Token> tokens, int startIndex) {

    TokenMatch* match = new TokenMatch();
    bool isMatch = true, matched = false;
    int i = startIndex;

    Production* prod = this->prod->GetLDO()->findProdById(source);
    if (prod != NULL) {
        TokenMatch* prodMatch = prod->GetRootProductionSet()->MatchStrict(tokens, i);
        if (prodMatch != NULL) {
            if (prodMatch->length > 0) {
                i += prodMatch->length;
                match->submatches.push_back(prodMatch);
            }
            matched = true;
        }
    }

    isMatch = multiplicity != "" || matched;

    if (!isMatch)
        return NULL;

    match->begin = startIndex;
    match->end = i;
    match->length = match->end - match->begin;
    match->match = vector<Token>(&tokens[match->begin], &tokens[match->end]);

    return match;
}

Markup* TokenMatch::GenerateMarkup(Markup* parent, bool addChildrenToParent) {
    Markup* r = NULL;
    if (addChildrenToParent) {
        if (parent != NULL)
            r = parent;
        else
            r = new Markup(prod);
    } else {
        r = new Markup(prod);
        if (parent != NULL)
            parent->AddChild(r);
    }

    string currentData;
    vector<TokenMatch*> sms = submatches;

    for (int i = 0; i <= length; i++) {
        Markup* c = NULL;
        TokenMatch* sub = NULL;

        for (int j = 0; j < sms.size(); j++) {
            if (sms[j]->begin == i + begin) {
                sub = sms[j];
                if (sub->isAction) {
                    sms.erase(sms.begin() + j);
                    break;
                }
            }
        }

        if (sub != NULL) {
            if (!sub->isAction) {
                c = sub->GenerateMarkup(r, sub->prod == "");
                i += sub->length - 1;
            } else {
                ActionRoutines::ExecuteAction(sub->prod, r);
                i--;
            }
        } else if (i < length) {
            c = new Markup(match[i].id, match[i].value);
            r->AddChild(c);
        }

        if (c != NULL) {
            if (currentData != "")
                currentData += " ";
            currentData += c->GetData();
        }
    }
    // r->SetData(currentData);

    return r;
}

```

```
void TokenMatch::Print(int tab) {
    if (prod != "") {
        for (int p = 0; p < tab; p++)
            cout << "\t";
        cout << prod << endl;
        tab++;
    }

    for (int i = 0; i < length; i++) {
        TokenMatch* sub = NULL;

        for (int j = 0; j < submatches.size(); j++) {
            if (submatches[j]->begin == i + begin) {
                sub = submatches[j];
                break;
            }
        }

        if (sub != NULL) {
            sub->Print(tab);
            i += sub->length - 1;
        } else {
            for (int p = 0; p < tab; p++)
                cout << "\t";
            cout << match[i].id << ": " << match[i].value << endl;
        }
    }
}

unordered_map<string, ActionRoutine*> ActionRoutines::actions = {
    { "DeclareVar", new DeclareVarAction() },
    { "AssignVar", new AssignVarAction() },
    { "ResolveExpr", new ResolveExprAction() },
    { "AccumulateVar", new AccumulateVarAction() }
};

Markup* ActionRoutines::ExecuteAction(string source, Markup* container) {
    regex r = regex("^[ \t]*([a-zA-Z_][a-zA-Z_0-9]*)[ \t]*(?:\\((.*)\\))?[ \t]*$");
    smatch matches;

    regex_search(source, matches, r);
    string actionID = matches[1].str();
    string actionParameters = matches[2].str();

    vector<Markup*> params = ResolveParameters(actionParameters, container);
    return ExecuteAction(actionID, container, params);
}
Markup* ActionRoutines::ExecuteAction(string actionID, Markup* container, vector<Markup*> params) {

    ActionRoutine* action = NULL;
    // cout << "Executed action " << actionID << endl;

    if ((action = ActionRoutines::actions[actionID]) != NULL) {
        return action->Execute(container, params);
    }

    return NULL;
}
vector<Markup*> ActionRoutines::ResolveParameters(string args, Markup* current) {
    vector<Markup*> params;

    if (args != "") {
        int groupLevel = 0;

        string arg = "";
        for (int i = 0; i < args.size(); i++) {
            if (args[i] == ',' && groupLevel == 0) {
                Markup* a = ResolveParameter(arg, current);
                params.push_back(a);
                arg = "";
            } else {
                if (args[i] == '(')
                    groupLevel++;
                else if (args[i] == ')')
                    groupLevel--;
                arg += args[i];
            }
        }
        if (arg != "") {
            Markup* a = ResolveParameter(arg, current);
            params.push_back(a);
        }

    }

    return params;
}
Markup* ActionRoutines::ResolveParameter(string arg, Markup* current) {
    regex fn = regex("^[ \t]*([a-zA-Z_][a-zA-Z_0-9]*)[ \t]*(\\(.*\\))?[ \t]*$");
    smatch matches;

    // cout << "Arg: " << arg << endl;

    if (regex_search(arg, matches, fn)) {
        string data = matches[0].str();
        return ExecuteAction(data, current);
    } else {

        int srcIndex = 0;
        string subscript = "";
        bool readSubscript = false;
        bool readAncestor = false;

        regex indexReg = regex("^(\\+|\\-)?\\d+$");
        regex sibOffsetReg = regex("^@((?:\\+|\\-)\\d+)$");
        regex keyReg = regex("^(v)?\"(.*)\"$");
        regex ancestorReg = regex("^\"(.*)\"$");

        for (int i = 0; i < arg.size() && current != NULL; i++) {
            if (readSubscript) {
                if (arg[i] == ']') {
                    readSubscript = false;
                    if (regex_search(subscript, matches, indexReg)) {
                        string index = matches[0].str();
                        subscript = "";
                        int n;
                        istringstream(index) >> n;
                        current = current->ChildAt(n);
                        srcIndex = current->IndexInParent();
                    } else if (regex_search(subscript, matches, keyReg)) {
                        bool dive = matches[1].str() != "";
                        string id = matches[2].str();
                        subscript = "";
                        if (dive)
                            current = current->FindFirstById(id);
                        else
                            current = current->FindFirstChildById(id);
                        srcIndex = current->IndexInParent();
                    } else if (regex_search(subscript, matches, sibOffsetReg)) {
                        string index = matches[1].str();
                        subscript = "";
                        int n;
                        istringstream(index) >> n;
                        n = srcIndex + n;
                        current = current->ChildAt(n);
                        srcIndex = current->IndexInParent();
                    } else {
                        cout << "Error parsing action routine parameter\n";
                        subscript = "";
                        break;
                    }

                } else {
                    subscript += arg[i];
                }
            } else if (readAncestor) {
                if (arg[i] == ')') {
                    readAncestor = false;
                    if (regex_search(subscript, matches, ancestorReg)) {
                        string ancestor = matches[1].str();
                        subscript = "";

                        int tempSrc;
                        Markup* temp = current;
                        do {
                            tempSrc = temp->IndexInParent();
                            temp = temp->Parent();
                        } while (temp != NULL && temp->GetID() != ancestor);

                        if (temp != NULL) {
                            srcIndex = tempSrc;
                            current = temp;
                        } else {
                            cout << "Error parsing action routine parameter - Production '" << ancestor << "' not found as an ancestor to the current node.\n";
                            break;
                        }
                    } else {
                        cout << "Error parsing action routine parameter\n";
                        subscript = "";
                        break;
                    }

                } else {
                    subscript += arg[i];
                }
            } else {
                if (arg[i] == '^') {
                    srcIndex = current->IndexInParent();
                    current = current->Parent();
                } else if (arg[i] == '[') {
                    readSubscript = true;
                } else if (arg[i] == '(') {
                    readAncestor = true;
                }
            }
        }
    }

    return current;

}
Markup* DeclareVarAction::Execute(Markup* container, vector<Markup*> params) {
    if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
        // don't do anything with the expression for now
        // this should be revised, but the incrementation screws with the Analyze module
        return NULL;
    }
    if (params.size() >= 2 && params[0] != NULL && params[1] != NULL) {
        string id = params[0]->GetData();
        string type = params[1]->GetData();
        Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
        if (statement == NULL)
            statement = container->FindAncestorById("function-definition");

        if (statement != NULL) {
            statement->localDeclarations[id] = type;
            cout << "Declared " << id << " with type " << type << endl;
        }
    } else {
        cout << "Failed to read variable declaration\n";
    }
    return NULL;
}
Markup* AssignVarAction::Execute(Markup* container, vector<Markup*> params) {
    if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
        // don't do anything with the expression for now
        // this should be revised, but the incrementation screws with the Analyze module
        return NULL;
    }
    if (params.size() >= 2 && params[0] != NULL && params[1] != NULL) {
        string id = params[0]->GetData();
        Markup* value = params[1];
        Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
        if (statement == NULL)
            statement = container->FindAncestorById("function-definition");

        if (statement != NULL) {
            statement->localValues[id] = value;
            cout << "Assigned " << id << " a value of " << value->GetData() << endl;
        }
    } else {
        cout << "Failed to read assignment\n";
    }
    return NULL;
}
Markup* AccumulateVarAction::Execute(Markup* container, vector<Markup*> params) {
    if (container->FindAncestorById("for-increment") != NULL || container->FindAncestorById("for-init") != NULL) {
        // don't do anything with the expression for now
        // this should be revised, but the incrementation screws with the Analyze module
        return NULL;
    }
    if (params.size() >= 3 && params[0] != NULL && params[1] != NULL && params[2] != NULL) {
        Markup* ident = params[1]->FindFirstById("identifier");
        if (ident != NULL) {
            string id = ident->GetData();
            Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
            if (statement == NULL)
                statement = container->FindAncestorById("function-definition");

            if (statement != NULL) {
                Markup* data = new Markup("algebraic-expression");
                data->localDeclarations = container->AccessibleDeclarations();
                data->localValues = container->AccessibleValues();

                data->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { ident }));
                Markup* tail = new Markup("algebraic-expression-tail");
                Markup* expr = new Markup("operation-expression");
                string opVal = "";
                string assignOp = params[0]->GetID();
                string assignData = params[0]->GetData().substr(0, 1);
                if (assignOp == "PLUS_ASSIGN")
                    opVal = "PLUS";
                else if (assignOp == "MINUS_ASSIGN")
                    opVal = "MINUS";
                else if (assignOp == "ASTERISK_ASSIGN")
                    opVal = "ASTERISK";
                else if (assignOp == "SLASH_ASSIGN")
                    opVal = "SLASH";
                Markup* op = new Markup("math-binary-op");
                // TODO this won't work if the particular language doesn't have shorthand assignments like this
                op->AddChild(new Markup(opVal, assignData));

                expr->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { params[2] }));
                tail->AddChild(op);
                tail->AddChild(expr);
                data->AddChild(tail);
                Markup* value = ActionRoutines::ExecuteAction("ResolveExpr", container, { data });

                statement->localValues[id] = value;
                cout << "Assigned " << id << " a value of " << value->GetData() << endl;
            }
        }
    } else if(params.size() == 2 && params[0] != NULL && params[1] != NULL) {
        Markup* ident = params[1];
        Markup* uop = params[0]->ChildAt(0);

        string id = ident->GetData();
        Markup* statement = container->GetID() == "statement" || container->GetID() == "function-definition" ? container : container->FindAncestorById("statement");
        if (statement == NULL)
            statement = container->FindAncestorById("function-definition");

        if (statement != NULL) {
            Markup* data = new Markup("algebraic-expression");
            data->localDeclarations = container->AccessibleDeclarations();
            data->localValues = container->AccessibleValues();

            data->AddChild(ActionRoutines::ExecuteAction("ResolveExpr", container, { ident }));
            Markup* tail = new Markup("algebraic-expression-tail");
            Markup* expr = new Markup("operation-expression");
            Markup* op = NULL;
            if (uop->GetID() == "INCR") {
                op = new Markup("PLUS", "+");
            } else if (uop->GetID() == "DECR") {
                op = new Markup("MINUS", "-");
            }
            Markup* binaryOp = new Markup("math-binary-op");
            binaryOp->AddChild(op);
            tail->AddChild(binaryOp);
            expr->AddChild(new Markup("INT_LITERAL", "1"));
            tail->AddChild(expr);
            data->AddChild(tail);
            Markup* value = ActionRoutines::ExecuteAction("ResolveExpr", container, { data });

            statement->localValues[id] = value;
            cout << "Assigned " << id << " a value of " << value->GetData() << endl;
        }
    } else {
        cout << "Failed to accumulate\n";
    }
    return NULL;
}
Markup* ResolveExprAction::Execute(Markup* container, vector<Markup*> params) {
    if (params.size() >= 1 && params[0] != NULL) {
        return ResolveExpr(params[0]);
    } else {
        cout << "Failed to resolve expression\n";
    }
    return NULL;
}
Markup* ResolveExprAction::ResolveExpr(Markup* data) {
    string id = data->GetID();
    // <grouped-expression>|<method-invocation>|<assignment>|<operation>|<simple-expression>

    if (data->FindAncestorById("for-increment") != NULL || data->FindAncestorById("for-init") != NULL) {
        // don't do anything with the expression for now
        // this should be revised, but the incrementation screws with the Analyze module
    } else if (id == "assign-expression") {
        data = ResolveExpr(data->ChildAt(0));
    } if (id == "grouped-expression") {
        data = ResolveExpr(data->FindFirstChildById("expression")->ChildAt(0));
    } else if (id == "operation-expression") {
        data = ResolveExpr(data->ChildAt(0));
    } else if (id == "simple-expression") {
        data = ResolveExpr(data->ChildAt(0));
        // <member-access>|<subscript-access>|<literal>|<identifier>
        // TODO member-access & subscript-access?
    } else if (id == "literal") {
        data = data->ChildAt(0);
        // <bool-literal>|[FLOAT_LITERAL]|[INT_LITERAL]|[STRING_LITERAL]
    } else if (id == "identifier" || id == "ID") {
        unordered_map<string, Markup*> assignments = data->AccessibleValues();
        string var = data->GetData();
        if (assignments[var] != NULL) {
            data = assignments[var];
        } else {
            cout << "Variable " << var << " may be unassigned\n";
        }
    } else if (id == "operation") {
        data = ResolveExpr(data->ChildAt(0));
        //<binary-expression>|<unary-expression>
    } else if (id == "unary-expression") {
        // TODO
    } else if (id == "binary-expression") {
        data = ResolveExpr(data->ChildAt(0));
        //<relational-expression>|<algebraic-expression>|<logical-expression>
    } else if (id == "algebraic-expression") {
        vector<Markup*> operands = { ResolveExpr(data->ChildAt(0)) };
        vector<Markup*> operators;
        vector<Markup*> tails = data->FindFirstChildById("algebraic-expression-tail")->RecursiveElements();
        for (int i = 0; i < tails.size(); i++) {
            operators.push_back(tails[i]->FindFirstChildById("math-binary-op")->ChildAt(0));
            operands.push_back(ResolveExpr(tails[i]->FindFirstChildById("operation-expression")->ChildAt(0)));
        }

        // Process multiplication and division
        for (int i = operators.size() - 1; i >= 0; i--) {
            Markup* op2 = operands[i + 1];
            Markup* op1 = operands[i];
            operands.erase(operands.begin() + i, operands.begin() + i + 2);
            Markup* op = operators[i];
            operators.erase(operators.begin() + i);
            string opId = op->GetID();
            // did both operands resolve to int literals
            if (op1->GetID() == "INT_LITERAL" && op2->GetID() == "INT_LITERAL" && (opId == "ASTERISK" || opId == "SLASH")) {
                long op1data, op2data, result;
                istringstream(op1->GetData()) >> op1data;
                istringstream(op2->GetData()) >> op2data;
                if (opId == "ASTERISK")
                    result = op1data * op2data;
                else if (opId == "SLASH")
                    result = op1data / op2data;
                Markup* r = new Markup("INT_LITERAL", to_string(result));
                operands.insert(operands.begin() + i, r);
            } else {
                operators.insert(operators.begin() + i, op);
                operands.insert(operands.begin() + i, op2);
                operands.insert(operands.begin() + i, op1);
            }
        }
        // process addition and subtraction
        for (int i = operators.size() - 1; i >= 0; i--) {
```

```cpp
            Markup* op2 = operands[i + 1];
            Markup* op1 = operands[i];
            operands.erase(operands.begin() + i, operands.begin() + i + 2);
            Markup* op = operators[i];
            operators.erase(operators.begin() + i);
            string opId = op->GetID();
            // did both operands resolve to int literals?
            if (op1->GetID() == "INT_LITERAL" && op2->GetID() == "INT_LITERAL" && (opId == "PLUS" || opId == "MINUS")) {
                long op1data, op2data, result;
                istringstream(op1->GetData()) >> op1data;
                istringstream(op2->GetData()) >> op2data;
                if (opId == "MINUS")
                    op2data *= -1;
                result = op1data + op2data;
                Markup* r = new Markup("INT_LITERAL", to_string(result));
                operands.insert(operands.begin() + i, r);
            } else {
                operators.insert(operators.begin() + i, op->Clone());
                operands.insert(operands.begin() + i, op2->Clone());
                operands.insert(operands.begin() + i, op1->Clone());
            }
        }

        if (operands.size() == 1) {
            data = operands[0];
        } else {
            data = new Markup("generated-expression");
            int i;
            for (i = 0; i < operators.size(); i++) {
                data->AddChild(operands[i]);
                data->AddChild(operators[i]);
            }
            data->AddChild(operands[i]);
        }

    }

    return data;
}
```