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

#include "ControlModule.h"

ControlModule::ControlModule() {

}

ControlModule::~ControlModule() {

}

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

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

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

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

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

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

    FormatOutput();
}

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

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

```

```cpp
LANGUAGE_DESCRIPTOR_OBJECT ControlModule::ReadLanguageFile(SOURCE_LANGUAGE sourceLanguage) throw (std::string) {
    LANGUAGE_DESCRIPTOR_OBJECT languageDescriptor;
    try {
        // read and parse file;
        languageDescriptor = new LanguageDescriptorObject(sourceLanguage);
    } catch (...) {

        throw "Language '" + sourceLanguage + "' could not be read";
    }

    return languageDescriptor;
}

CODE_OUTPUT ControlModule::CoalesceCode(CODE_INPUT codeSnippets) {
    CODE_OUTPUT code;

    // do some iterations over codeSnippets to unify it

    return code;
}

MARKUP_OBJECT ControlModule::Parse(CODE_OUTPUT code, LANGUAGE_DESCRIPTOR_OBJECT languageDescriptor) {
    MARKUP_OBJECT markup = new Markup("ROOT");

    vector<Token> tokens = languageDescriptor->Tokenize(code[0]);

    // for (int i = 0; i < tokens.size(); i++) {
    //     cout << "State machine accepted token '" << tokens[i].id << "' with data '" << tokens[i].value << "'\n";
    // }

    vector<Production*> prods = languageDescriptor->GetProductions();
    vector<vector<Token>> tokenSets;
    tokenSets.push_back(tokens);

    bool matched = false;

    for (int j = 0; j < tokenSets.size(); j++) {
        matched = false;

        for (int p = 0; p < tokenSets[j].size() && !matched; p++) {

            for (int i = 0; i < prods.size() && !matched; i++) {
                TokenMatch* match = prods[i]->MatchStrict(tokenSets[j]);
                if (match != NULL) {
                    vector<Token> s = vector<Token>(tokenSets[j].begin(), tokenSets[j].begin() + match->begin);
                    vector<Token> e = vector<Token>(tokenSets[j].begin() + match->end, tokenSets[j].begin() + tokenSets[j].size());

                    tokenSets.erase(tokenSets.begin() + j);

                    bool dec = false;
                    if (e.size() > 0) {
                        tokenSets.insert(tokenSets.begin() + j, e);
                        dec = true;
                    }
                    if (s.size() > 0) {
                        tokenSets.insert(tokenSets.begin() + j, s);
                        dec = true;
                    }
                    if (dec)
                        j--;
                    // cout << "MATCHED: " << prods[i]->GetId() << ", start = " << match->begin << ", end = " << match->end << ", length = " << match->length << endl;
                    // match->Print(0);
                    Markup* m = match->GenerateMarkup();
                    markup->AddChild(m);
                    // markupList.push_back(m);
                    matched = true;
                }
            }
        }
    }

    // markup->Print();

    return markup;
}

void ControlModule::Execute(MARKUP_OBJECT markup, LANGUAGE_DESCRIPTOR_OBJECT ldo, MODULE_ID moduleID, FUNCTION_ARGS functionArgs) {
    MODULE_REF ref = ModuleRetrieval(moduleID);
    if (ref != NULL)
        ModuleExecution(ref, markup, ldo, functionArgs);
}

MODULE_REF ControlModule::ModuleRetrieval(MODULE_ID moduleID) {

    try {
        return GetModule(moduleID);
    } catch (...) {
        returnData->AddStandardError("Module '" + moduleID + "' could not be found.");
        return NULL;
    }

}

void ControlModule::ModuleExecution(MODULE_REF moduleRef, MARKUP_OBJECT markup, LANGUAGE_DESCRIPTOR_OBJECT ldo, FUNCTION_ARGS functionArgs) {
    try {
        // attempt to execute the module
        returnData = moduleRef->Execute(markup, ldo, functionArgs, returnData);
    } catch (...) {
        returnData->AddStandardError("An error occurred while trying to execute the module!");
    }
}

void ControlModule::FormatOutput() {

    cout << "CASP_RETURN_DATA_START\n";
    returnData->Print();
    cout << "\nCASP_RETURN_DATA_END\n";

}
```