```cpp
#include "TranslateModule.h"

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

TranslateModule::TranslateModule() {}

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

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

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

    if (targetLanguage != "") {

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

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

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

    return returnData;

}

void TranslateModule::Translate(Markup* markup) {

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

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

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

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

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

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

    returnData->Data()->Add("TranslatedSource", CreateObject({
        { "Language" , CreateLeaf(target_ldo->GetLanguage()) },
        { "Data" , CreateLeaf(PrettyPrint(tl2)) }
    }));

}

string TranslateModule::PrettyPrint(vector<Token> tokens) {
    int i = 0;
    return PrintBlockBody(tokens, &i, 0);
}

string TranslateModule::PrintBlockBody(vector<Token> tokens, int* index, int tabIndex) {
    string str = "";
    int i;
    int size = tokens.size();
    bool finishedBlock = false;
    bool endStmt = false;
    bool forStmts = false;

    for (i = *index; i < size; i++) {
        Token t = tokens[i];

        if (t.id == "L_CU_BRACKET") {
            if (i != 0)
                str += "\n";
            str += Helpers::DupStr("    ", tabIndex);
            str += t.value;
            str += "\n" + Helpers::DupStr("    ", tabIndex + 1);
            i++;
            str += PrintBlockBody(tokens, &i, tabIndex + 1);
            i--;
            finishedBlock = true;
            endStmt = false;
            forStmts = false;
            continue;
        } else if (t.id == "R_CU_BRACKET") {
            if (!finishedBlock)
                break;
            str += "\n";
            str += Helpers::DupStr("    ", tabIndex);
            str += t.value;
            str += "\n";
            str += Helpers::DupStr("    ", tabIndex);
            endStmt = false;
        } else if (t.id == "SEMICOLON" && !forStmts) {
            str += t.value;
            endStmt = true;
        } else {
            if (endStmt) {
                str += "\n" + Helpers::DupStr("    ", tabIndex);
                endStmt = false;
            }
            if (t.id == "FOR")
                forStmts = true;
            str += t.value + " ";
        }
        finishedBlock = false;
    }
    *index = i;
    return str;
}


Markup* TranslateModule::MatchTargetProd(Markup* markup) {

    string nodeId = markup->GetID();

    if (!markup->IsLeaf()) {
        Production* targetProd = target_ldo->findProdById(nodeId);

        if (targetProd != NULL) {
            return TranslateProd(markup, targetProd);
        } else {
            Markup* ret = new Markup(nodeId);
            Markup* t = NULL;
            returnData->AddStandardWarning("No matching translation for construct '" + nodeId + "'");
            // add warning that this node could not be translated
            // vector<Markup*> children = markup->Children();
            // for (int i = 0; i < children.size(); i++) {
            //     t = MatchTargetProd(children[i]);
            //     if (t != NULL) {
            //         ret->AddChild(t);
            //     }
            // }
            // return ret;
            return NULL;
        }
    } else {
        string nodeValue = markup->GetData();
        bool dynamicTerminal = source_ldo->LookupTerminalValue(nodeId) == "";
        if (!dynamicTerminal) {
            string newTerminal = target_ldo->LookupTerminalValue(nodeId);
            if (newTerminal != "") {
                return new Markup(nodeId, newTerminal);
            } else {
                returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
                // add warning that there is no translation
                return NULL;//new Markup(nodeId, nodeValue);
            }
        } else {
            returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
            // add warning that this cannot be translated
            return NULL;// new Markup(nodeId, nodeValue);
        }

    }

    return NULL;
}

Markup* TranslateModule::TranslateProd(Markup* source, Production* target) {
    Markup* newMarkup = new Markup(target->GetId());
    vector<ProductionSet*> children = target->GetRootProductionSet()->GetChildren();

    for (int i = 0; i < children.size(); i++) {
        ProductionSet* p = children[i];
        Markup* c = NULL;

        switch (p->GetType()) {
            case _Terminal:
                c = HandleTerminal(source, p, true);
                break;
            case _Production:
                c = HandleProduction(source, p, true);
                break;
            case _Alternation:
                c = HandleAlternation(source, p);
                break;
        }

        if (c != NULL) {
            newMarkup->AddChild(c);
        }
    }

    return newMarkup;
}
Markup* TranslateModule::HandleTerminal(Markup* source, ProductionSet* set, bool fillInOnNoMatch) {
    string nodeId = set->GetSource();
    Markup* sourceTerminal = source->FindFirstChildById(nodeId);
    string newTerminal = target_ldo->LookupTerminalValue(nodeId);

    if (fillInOnNoMatch || sourceTerminal != NULL) {
        if (newTerminal != "") {
            return new Markup(nodeId, newTerminal);
        } else if (sourceTerminal != NULL) {
            // returnData->AddStandardWarning("The translation for terminal '" + nodeId + "' (value = '" + sourceTerminal->GetData() + "') is not guaranteed. Check syntax.");
            // add warning that this is an inconclusive translation
            return new Markup(nodeId, sourceTerminal->GetData());
        }
    }
    // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
    // add warning that there is no matching translation
    return NULL;
}
Markup* TranslateModule::HandleProduction(Markup* source, ProductionSet* set, bool dummyOnFail) {
    string nodeId = set->GetSource();
    Markup* ret = NULL;

    Markup* sourceProduction = source->FindFirstChildById(nodeId);
    if (sourceProduction != NULL) {
        Production* targetProd = target_ldo->findProdById(nodeId);
        ret = TranslateProd(sourceProduction, targetProd);
        if (ret != NULL) {
            return ret;
        }
    } else {

    }
    if (dummyOnFail && set->GetMultiplicity() != "?")
        ret = new Markup(nodeId, "<" + nodeId + ">");
    // returnData->AddStandardWarning("No matching translation for production '" + nodeId + "'");
    // add warning that there is no matching translation
    return ret;
}
Markup* TranslateModule::HandleAlternation(Markup* source, ProductionSet* set) {
    Markup* newMarkup = NULL;
    vector<ProductionSet*> children = set->GetChildren();

    for (int i = 0; i < children.size() && newMarkup == NULL; i++) {
        ProductionSet* p = children[i];

        switch (p->GetType()) {
            case _Terminal:
                newMarkup = HandleTerminal(source, p, false);
                break;
            case _Production:
                newMarkup = HandleProduction(source, p, false);
                break;
            case _Alternation:
                newMarkup = HandleAlternation(source, p);
                break;
        }
    }

    if (newMarkup == NULL) {
        // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
    }

    return newMarkup;
```

```cpp
}


void TranslateModule::ReadLanguageFile(string targetLanguage) {
    // read and parse file;
    target_ldo = new LanguageDescriptorObject(targetLanguage);
}
```