

```
1 #include "TranslateModule.h"
2
3 static string _TranslateModule = RegisterPlugin("Translate", new TranslateModule());
4
5 TranslateModule::TranslateModule() {}
6
7 CASP_Return* TranslateModule::Execute(Markup* markup, LanguageDescriptorObject* source_ldo, vector<arg> fnArgs, CASP_Return* inputReturn) {
8     returnData = (inputReturn != NULL ? inputReturn : new CASP_Return());
9
10    // cout << "This is the entry point for the " << _TranslateModule << " Module!\n";
11
12    bool languageRead = true;
13    this->source_ldo = source_ldo;
14    string targetLanguage = Helpers::ParseArgument("targetlang", fnArgs);
15
16    if (targetLanguage != "") {
17
18        try {
19            ReadLanguageFile(targetLanguage);
20        } catch (...) {
21            returnData->AddStandardError("Language '" + targetLanguage + "' could not be read. Could not proceed with translation.");
22            languageRead = false;
23        }
24
25        if (languageRead) {
26            try {
27                Translate(markup);
28            } catch (...) {
29                returnData->AddStandardError("Error while processing translation.");
30            }
31        }
32
33    } else {
34        returnData->AddStandardError("Target language not provided. Make sure to use argument 'targetlang'.");
35    }
36
37    return returnData;
38 }
39
40 void TranslateModule::Translate(Markup* markup) {
41
42     Markup* targetRoot = new Markup("ROOT");
43     string nodeId = markup->GetID();
44
45     vector<Markup*> children = markup->Children();
46     for (int i = 0; i < children.size(); i++) {
47         Markup* c = MatchTargetProd(children[i]);
48         if (c != NULL)
49             targetRoot->AddChild(c);
50     }
51
52     // markup->Print();
53     // targetRoot->Print();
54
55     vector<Token> t11 = source_ldo->Tokenize(markup);
56     vector<Token> t12 = target_ldo->Tokenize(targetRoot);
57
58     cout << endl << PrettyPrint(t11) << endl << endl;
59     cout << endl << PrettyPrint(t12) << endl << endl;
60
61     returnData->Data()->Add("OriginalSource", CreateObject({
62         { "Language", CreateLeaf(source_ldo->GetLanguage()) },
63         { "Data", CreateLeaf(PrettyPrint(t11)) }
64     }));
65 }
66
```

```

67     returnData->Data()->Add("TranslatedSource", CreateObject({
68         { "Language" , CreateLeaf(target_ldo->GetLanguage()) },
69         { "Data" , CreateLeaf(PrettyPrint(t12)) }
70     }));
71 }
72 }
73
74 string TranslateModule::PrettyPrint(vector<Token> tokens) {
75     int i = 0;
76     return PrintBlockBody(tokens, &i, 0);
77 }
78
79 string TranslateModule::PrintBlockBody(vector<Token> tokens, int* index, int tabIndex) {
80     string str = "";
81     int i;
82     int size = tokens.size();
83     bool finishedBlock = false;
84     bool endStmt = false;
85     bool forStmts = false;
86
87     for (i = *index; i < size; i++) {
88         Token t = tokens[i];
89
90         if (t.id == "L_CU_BRACKET") {
91             if (i != 0)
92                 str += "\n";
93             str += Helpers::DupStr("    ", tabIndex);
94             str += t.value;
95             str += "\n" + Helpers::DupStr("    ", tabIndex + 1);
96             i++;
97             str += PrintBlockBody(tokens, &i, tabIndex + 1);
98             i--;
99             finishedBlock = true;
100             endStmt = false;
101             forStmts = false;
102             continue;
103         } else if (t.id == "R_CU_BRACKET") {
104             if (!finishedBlock)
105                 break;
106             str += "\n";
107             str += Helpers::DupStr("    ", tabIndex);
108             str += t.value;
109             str += "\n";
110             str += Helpers::DupStr("    ", tabIndex);
111             endStmt = false;
112         } else if (t.id == "SEMICOLON" && !forStmts) {
113             str += t.value;
114             endStmt = true;
115         } else {
116             if (endStmt) {
117                 str += "\n" + Helpers::DupStr("    ", tabIndex);
118                 endStmt = false;
119             }
120             if (t.id == "FOR")
121                 forStmts = true;
122             str += t.value + " ";
123         }
124         finishedBlock = false;
125     }
126     *index = i;
127     return str;
128 }
129
130
131 Markup* TranslateModule::MatchTargetProd(Markup* markup) {
132
133     string nodeId = markup->GetID();

```

```

134
135 if (!markup->IsLeaf()) {
136     Production* targetProd = target_ldo->findProdById(nodeId);
137
138     if (targetProd != NULL) {
139         return TranslateProd(markup, targetProd);
140     } else {
141         Markup* ret = new Markup(nodeId);
142         Markup* t = NULL;
143         returnData->AddStandardWarning("No matching translation for construct '" + nodeId + "'");
144         // add warning that this node could not be translated
145         // vector<Markup*> children = markup->Children();
146         // for (int i = 0; i < children.size(); i++) {
147         //     t = MatchTargetProd(children[i]);
148         //     if (t != NULL) {
149         //         ret->AddChild(t);
150         //     }
151         // }
152         // return ret;
153         return NULL;
154     }
155 } else {
156     string nodeValue = markup->GetData();
157     bool dynamicTerminal = source_ldo->LookupTerminalValue(nodeId) == "";
158     if (!dynamicTerminal) {
159         string newTerminal = target_ldo->LookupTerminalValue(nodeId);
160         if (newTerminal != "") {
161             return new Markup(nodeId, newTerminal);
162         } else {
163             returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
164             // add warning that there is no translation
165             return NULL; // new Markup(nodeId, nodeValue);
166         }
167     } else {
168         returnData->AddStandardWarning("No translation for terminal '" + nodeId + "'");
169         // add warning that this cannot be translated
170         return NULL; // new Markup(nodeId, nodeValue);
171     }
172 }
173 }
174
175 return NULL;
176 }
177
178 Markup* TranslateModule::TranslateProd(Markup* source, Production* target) {
179     Markup* newMarkup = new Markup(target->GetId());
180     vector<ProductionSet*> children = target->GetRootProductionSet()->GetChildren();
181
182     for (int i = 0; i < children.size(); i++) {
183         ProductionSet* p = children[i];
184         Markup* c = NULL;
185
186         switch (p->GetType()) {
187             case _Terminal:
188                 c = HandleTerminal(source, p, true);
189                 break;
190             case _Production:
191                 c = HandleProduction(source, p, true);
192                 break;
193             case _Alternation:
194                 c = HandleAlternation(source, p);
195                 break;
196         }
197
198         if (c != NULL) {
199             newMarkup->AddChild(c);
200         }

```

```
201 }
202
203 return newMarkup;
204 }
205 Markup* TranslateModule::HandleTerminal(Markup* source, ProductionSet* set, bool fillInOnNoMatch) {
206     string nodeId = set->GetSource();
207     Markup* sourceTerminal = source->FindFirstChildById(nodeId);
208     string newTerminal = target_ldo->LookupTerminalValue(nodeId);
209
210     if (fillInOnNoMatch || sourceTerminal != NULL) {
211         if (newTerminal != "") {
212             return new Markup(nodeId, newTerminal);
213         } else if (sourceTerminal != NULL) {
214             // returnData->AddStandardWarning("The translation for terminal '" + nodeId + "' (value = '" + sourceTerminal->GetData() + "') is not guaranteed. Check syntax.");
215             // add warning that this is an inconclusive translation
216             return new Markup(nodeId, sourceTerminal->GetData());
217         }
218     }
219     // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
220     // add warning that there is no matching translation
221     return NULL;
222 }
223 Markup* TranslateModule::HandleProduction(Markup* source, ProductionSet* set, bool dummyOnFail) {
224     string nodeId = set->GetSource();
225     Markup* ret = NULL;
226
227     Markup* sourceProduction = source->FindFirstChildById(nodeId);
228     if (sourceProduction != NULL) {
229         Production* targetProd = target_ldo->findProdById(nodeId);
230         ret = TranslateProd(sourceProduction, targetProd);
231         if (ret != NULL) {
232             return ret;
233         }
234     } else {
235
236     }
237     if (dummyOnFail && set->GetMultiplicity() != "?")
238         ret = new Markup(nodeId, "<" + nodeId + ">");
239     // returnData->AddStandardWarning("No matching translation for production '" + nodeId + "'");
240     // add warning that there is no matching translation
241     return ret;
242 }
243 Markup* TranslateModule::HandleAlternation(Markup* source, ProductionSet* set) {
244     Markup* newMarkup = NULL;
245     vector<ProductionSet*> children = set->GetChildren();
246
247     for (int i = 0; i < children.size() && newMarkup == NULL; i++) {
248         ProductionSet* p = children[i];
249
250         switch (p->GetType()) {
251             case _Terminal:
252                 newMarkup = HandleTerminal(source, p, false);
253                 break;
254             case _Production:
255                 newMarkup = HandleProduction(source, p, false);
256                 break;
257             case _Alternation:
258                 newMarkup = HandleAlternation(source, p);
259                 break;
260         }
261     }
262
263     if (newMarkup == NULL) {
264         // returnData->AddStandardWarning("No matching translation for terminal '" + nodeId + "'");
265     }
266
267     return newMarkup;
```

```
268 }
269
270
271 void TranslateModule::ReadLanguageFile(string targetLanguage) {
272     // read and parse file;
273     target_ldo = new LanguageDescriptorObject(targetLanguage);
274 }
```