```cpp
//#include "splwpch.h"
#include "Log.h"

#include "spdlog/sinks/stdout_color_sinks.h"

namespace Spliwaca
{

	std::shared_ptr<spdlog::logger> Log::s_CoreLogger;
	std::shared_ptr<spdlog::logger> Log::s_ClientLogger;

	void Log::Init() {
		spdlog::set_pattern("%^[%H:%M:%S:%f] %n: %v%$");

		s_CoreLogger = spdlog::stdout_color_mt("Supernova");
		s_CoreLogger->set_level(spdlog::level::trace);

		s_ClientLogger = spdlog::stdout_color_mt("App");
		s_ClientLogger->set_level(spdlog::level::trace);
	}
}
```

```
#include "Lexer.h"
#include <regex>
#include <fstream>
#include <iostream>
#include <map>
#include <algorithm>
//#include "Instrumentor.h"
#include "Log.h"
#include "UtilFunctions.h"

namespace Spliwaca
{
    bool itemInVect(const std::vector<std::string>& v, std::string t)
    {
        for (int i = 0; i < v.size(); i++)
        {
            if (v.at(i) == t)
            {
                return true;
            }
        }
        return false;
    }

    bool itemInVect(const std::vector<char>& v, char t)
    {
        for (int i = 0; i < v.size(); i++)
        {
            if (v.at(i) == t)
            {
                return true;
            }
        }
        return false;
    }

    std::shared_ptr<Lexer> Lexer::Create(std::string file)
    {
        return std::shared_ptr<Lexer>(new Lexer(file));
    }

    Lexer::Lexer(std::string fileLocation)
        : m_Tokens(new std::vector<std::shared_ptr<Token>>()), m_FileLocation(fileLocation)
    {
        SPLW_INFO("Beginning file open");
        std::ifstream file;
        file.open(m_FileLocation);
        //char* fileContents;

        if (file.is_open())
        {
            std::string line;
            while (std::getline(file, line))
            {
                m_FileString.append(line + "\n");
            }
            //SPLW_TRACE("File contents: {0}", m_FileString);
            //std::cout << fileContents << "\n";
        }
        else
        {
            SPLW_ERROR("Could not open source file: {0}", m_FileLocation);
            system("PAUSE");
            exit(2);
        }

        file.close();
        SPLW_INFO("Closed file");

        //std::shared_ptr<Token> token;
        //token.reset(new Token(TokenType::UnfinishedToken, fileContents.c_str(), 0, 0));

        //m_Tokens->push_back(token);
    }

    void Lexer::makeToken(std::string tokenContents)
    {
        if (tokenContents == "\x04")
        {
            //End of file. Final cleanup time, and return error if we are missing something.
            if (flags & 16)
            {
                SPLW_ERROR("Missing double quote at end of file.");
                return;
            }
            else if (flags & 8)
```

```
   87:             {
   88:                 SPLW_ERROR("Missing single quote at end of file.");
   89:                 return;
   90:             }
   91:             else if (flags & 4)
   92:             {
   93:                 flags &= 0b11111011;
   94:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Raw, persistent_contents.c_
str(), m_StoredLineNumber, m_StoredColumnNumber)));
   95:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Newline, "\n", m_LineNumber
, m_ColumnNumber)));
   96:                 return;
   97:             }
   98:             else if (flags & 2)
   99:             {
  100:                 if (tokenContents == "*/")
  101:                 {
  102:                     flags &= 0b11111101;
  103:                 }
  104:                 else
  105:                     SPLW_ERROR("Missing end of block comment at end of file.");
  106:                 return;
  107:             }
  108:             else if (flags & 1)
  109:             {
  110:                 flags &= 0b11111110;
  111:                 return;
  112:             }
  113:             m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::eof, "", m_LineNumber, m_Column
Number)));
  114:             return;
  115:         }/* ---------------------------------END CLEANUP-------------------------------- */
  116:         else if (flags == 0)
  117:         {
  118:             if (s_KeywordDict.find(tokenContents) != s_KeywordDict.end() && tokenContents != "RAW" && to
kenContents != "OUTPUT" && tokenContents != "/*" && tokenContents != "//")
  119:             {
  120:                 //We have a keyword!
  121:                 m_Tokens->push_back(std::make_shared<Token>(Token(s_KeywordDict.at(tokenContents), token
Contents.c_str(), m_LineNumber, m_ColumnNumber)));
  122:
  123:             }
  124:             else if (tokenContents == "/*")
  125:             {
  126:                 flags |= 0b00000010;
  127:             }
  128:             else if (tokenContents == "//")
  129:             {
  130:                 flags |= 0b00000001;
  131:             }
  132:             else if (tokenContents == "OUTPUT")
  133:             {
  134:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Output, tokenContents.c_str
(), m_LineNumber, m_ColumnNumber)));
  135:                 flags |= 0b00100100;
  136:                 m_StoredColumnNumber = m_ColumnNumber;
  137:                 m_StoredLineNumber = m_LineNumber;
  138:             }
  139:             else if (tokenContents == "RAW")
  140:             {
  141:                 flags |= 0b00100100;
  142:                 m_StoredColumnNumber = m_ColumnNumber;
  143:                 m_StoredLineNumber = m_LineNumber;
  144:             }
  145:             else if (tokenContents == "\"")
  146:             {
  147:                 flags |= 0b00010000;
  148:                 m_StoredColumnNumber = m_ColumnNumber;
  149:                 m_StoredLineNumber = m_LineNumber;
  150:             }
  151:             else if (tokenContents == "'")
  152:             {
  153:                 flags |= 0b00001000;
  154:                 m_StoredColumnNumber = m_ColumnNumber;
  155:                 m_StoredLineNumber = m_LineNumber;
  156:             }
  157:             else if (tokenContents == std::string(1, ' ') || tokenContents == "\t" || tokenContents == "
\f" || tokenContents == "")// || tokenContents == std::string(1, '\ufeff')) // Whitespace
  158:             {
  159:             }
  160:             else
  161:             {
  162:                 std::smatch m;
  163:                 //Use regexes
  164:                 if (std::regex_search(tokenContents, m, std::regex("(\\d|_)+(\\.\\d+)?i")) && m[0] == to
kenContents) // Matches complex regex
```

```
  165:                 {
  166:                     if (std::regex_search(tokenContents, m, std::regex("(\\d{1,3}(_\\d{3})+|\\d+)(\\.[0-
9]+)?i")) && m[0] != tokenContents)
  167:                         SPLW_WARN("Style Warning, line {0}, char {1}: Complex literals should have under
scores treated as commas.");
  168:                     m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Complex, tokenContents.
c_str(), m_LineNumber, m_ColumnNumber)));
  169:                 }
  170:                 else if (std::regex_search(tokenContents, m, std::regex("(\\d|_)+\\.\\d+")) && m[0] == t
okenContents) // Matches float regex
  171:                 {
  172:                     if (std::regex_search(tokenContents, m, std::regex("(\\d{1,3}(_\\d{3})+|\\d+)\\.[0-9
]+")) && m[0] != tokenContents)
  173:                         SPLW_WARN("Style Warning, line {0}, char {1}: Float literals should have undersc
ores treated as commas.");
  174:                     m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Float, tokenContents.c_
str(), m_LineNumber, m_ColumnNumber)));
  175:                 }
  176:                 else if (std::regex_search(tokenContents, m, std::regex("(\\d+_*)+")) && m[0] == tokenCo
ntents) // Matches int regex
  177:                 {
  178:                     if (std::regex_search(tokenContents, m, std::regex("\\d{1,3}(_\\d{3})+|\\d+")) && m[
0] != tokenContents)
  179:                         SPLW_WARN("Style Warning, line {0}, char {1}: Integer literals should have under
scores treated as commas.");
  180:                     m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Int, tokenContents.c_st
r(), m_LineNumber, m_ColumnNumber)));
  181:                 }
  182:                 else
  183:                 {
  184:                     char invalidChars[] = { '~', '\\', ';', '#', '$', '@', ''', ',', '?', '!', '%', '^',
 '<', '|', '\'', '&', ')', '*', '/', '+', '[', ']', '.', '"', '=', '{', '}', ':', '>', '(', '-'};
  185:                     bool valid = true;
  186:                     int index = 0;
  187:                     for (char c : tokenContents) {
  188:                         for (char d : invalidChars) {
  189:                             if (c == d) {
  190:                                 valid = false;
  191:                                 break;
  192:                             }
  193:                         }
  194:                         if (valid == false)
  195:                             break;
  196:                         index++;
  197:                     }
  198:                     if (valid)
  199:                         m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Identifier, tokenCo
ntents.c_str(), m_LineNumber, m_ColumnNumber)));
  200:                     else {
  201:                         //Error unexpected characters.
  202:                         SPLW_ERROR("Lexical Error: Unexpected character {0} in string: {1}", tokenConten
ts[index], tokenContents);
  203:                         RegisterLexicalError(0, m_LineNumber, m_ColumnNumber, tokenContents.size());
  204:                     }
  205:                 }
  206:             }
  207:         }
  208:         else if (flags & 32) // First char of RAW
  209:         {
  210:             flags &= 0b11011111;
  211:             if (tokenContents != " ")
  212:             {
  213:                 if (tokenContents == "\n")
  214:                 {
  215:                     flags &= 0b11111011;
  216:                     m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Raw, persistent_content
s.c_str(), m_StoredLineNumber, m_StoredColumnNumber)));
  217:                     m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Newline, "\n", m_LineNu
mber, m_ColumnNumber)));
  218:                     persistent_contents = "";
  219:                 }
  220:                 else
  221:                     persistent_contents.append(tokenContents);
  222:             }
  223:         }
  224:         else if (flags & 16) // Double quote
  225:         {
  226:             if (tokenContents == "\"")
  227:             {
  228:                 flags &= 0b11101111;
  229:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::String, persistent_contents
.c_str(), m_StoredLineNumber, m_StoredColumnNumber)));
  230:                 persistent_contents = "";
  231:             }
  232:             else
  233:                 persistent_contents.append(tokenContents);
```

```
234:         }
235:         else if (flags & 8) // Single quote
236:         {
237:             if (tokenContents == "'")
238:             {
239:                 flags &= 0b11110111;
240:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::String, persistent_contents
.c_str(), m_StoredLineNumber, m_StoredColumnNumber)));
241:                 persistent_contents = "";
242:             }
243:             else
244:                 persistent_contents.append(tokenContents);
245:         }
246:         else if (flags & 4) // Raw
247:         {
248:             if (tokenContents == "\n")
249:             {
250:                 flags &= 0b11111011;
251:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Raw, persistent_contents.c_
str(), m_StoredLineNumber, m_StoredColumnNumber)));
252:                 m_Tokens->push_back(std::make_shared<Token>(Token(TokenType::Newline, "\n", m_LineNumber
, m_ColumnNumber)));
253:                 persistent_contents = "";
254:             }
255:             else
256:                 persistent_contents.append(tokenContents);
257:         }
258:         else if (flags & 2) // Block comment
259:         {
260:             if (tokenContents == "*/")
261:             {
262:                 flags &= 0b11111101;
263:             }
264:         }
265:         else if (flags & 1)  // Comment
266:         {
267:             if (tokenContents == "\n")
268:             {
269:                 flags &= 0b11111110;
270:             }
271:         }
272:         else
273:         {
274:             SPLW_CRITICAL("WHY ARE WE IN THE ELSE? THIS SHOULD BE IMPOSSIBLE!!!!!!!!");
275:         }
276:         if (tokenContents == "\n")
277:         {
278:             m_LineNumber += 1;
279:             m_ColumnNumber = 0;
280:         }
281:         else
282:             m_ColumnNumber += tokenContents.size();
283:     }
284:
285:     std::shared_ptr<std::vector<std::shared_ptr<Token>>> Lexer::MakeTokens()
286:     {
287:         std::string s = m_FileString;
288:
289:         std::string intermediate = "";
290:         std::vector<char> splitChars = { ' ', '\n', '\t', '\f', '(', ')', '[', ']', '+', '-', '/', '*',
'!', '=', '%', '^', '&', '|', '<', '>', ',', '"', '\'', '.', ':' };
291:         std::vector<std::string> splitDuoStrings = { "/*", "*/", "//", "**", "??", "&&", "==", "!=", "<=
", ">=", "<<", ">>", "<-", "->", "||", "\\'", "\\\"" };
292:         std::vector<std::string> splitTrioStrings = { "=/=", "===" };
293:
294:         int i = 0;
295:         while (true)
296:         {
297:             //SPLW_INFO("Starting char {0}", s[i]);
298:             if ((s[i] & 0xffff) == 0xffef || (s[i] & 0xffff) == 0xffbb || (s[i] & 0xffff) == 0xffbf) {
299:                 i++;
300:                 continue;
301:             }
302:
303:             std::string c = std::string(1, s[i]);
304:             std::string duo = c; (i < s.size() - 1) ? duo += s[i + 1] : duo += "";
305:             std::string trio = duo; (i < s.size() - 2) ? trio += s[i + 2] : trio += "";
306:
307:             if (trio == "=/=")//itemInVect(splitTrioStrings, trio))
308:             {
309:                 if (intermediate != "")
310:                     makeToken(intermediate);
311:                 intermediate = c + s[i + 1] + s[i+2];
312:                 //intermediate += s[i + 1];
313:                 //intermediate += s[i + 2];
314:                 makeToken(intermediate);
```

```
315:                intermediate = "";
316:                i += 2;
317:            }
318:            else if (itemInVect(splitDuoStrings, duo))
319:            {
320:                if (intermediate != "")
321:                    makeToken(intermediate);
322:                intermediate = c + s[i+1];
323:                //intermediate += s[i + 1];
324:                makeToken(intermediate);
325:                intermediate = "";
326:                i++;
327:            }
328:            else if (itemInVect(splitChars, c[0]) || s_KeywordDict.find(c) != s_KeywordDict.end())
329:            {
330:                if (c != "." || charInStr(alphabetCharacters, intermediate[0]))
331:                {
332:                    if (intermediate != "")
333:                        makeToken(intermediate);
334:                    intermediate = c;
335:                    makeToken(intermediate);
336:                    intermediate = "";
337:                }
338:                else
339:                {
340:                    intermediate += c;
341:                }
342:            }
343:            else
344:            {
345:                intermediate += c;
346:            }
347:            i++;
348:            if (i >= s.size())
349:                break;
350:        }
351:        makeToken(intermediate);
352:        makeToken("\x004");
353:        return m_Tokens;
354:    }
355: }
```

```cpp
#include "Nodes.h"

namespace Spliwaca {
	std::string IdentNode::GetContents() {
		if (cachedContents != "")
			return cachedContents;
		std::string rv = "";
		//SPLW_INFO("{0}", ids.at(0)->GetContents());
		if (ids.at(0)->GetContents() != "_INTERPRETER") {
			rv += ids.at(0)->GetContents();
			for (size_t i = 1; i < ids.size(); i++) {
				rv += "." + ids.at(i)->GetContents();
			}
		} else if (ids.size() > 1) {
			rv += /*"__builtins__." +*/ ids.at(1)->GetContents();
			for (size_t i = 2; i < ids.size(); i++) {
				rv += "." + ids.at(i)->GetContents();
			}
		} else {
			return "_INTERPRETER";
		}
		cachedContents = rv;
		return rv;
	}

    std::string boolToString(bool Bool) {
        if (Bool) {
            return "True";
        }
        return "False";
    }

    std::string IdentNode::GenerateGetattrTree(ImportConfig *importConfig, bool &interpreter_var, bool minus_one) {
        if (cachedGetattrMinusOne != "" && minus_one)
            return cachedGetattrMinusOne;
        if (cachedGetattr != "")
            return cachedGetattr;

        if (ids.size() <= 1 && minus_one) {
            SPLW_CRITICAL("Attempting to get attribute of an identifier with no accesses. This is a bug.");
            return "";
        }
        if (ids.at(0)->GetContents() == "_INTERPRETER") {
            interpreter_var = true;
            return GetContents();
        }
        //SPLW_INFO("{0}, {1}, {2}, {3}", importConfig->allowImport, importConfig->allowPyImport, importConfig->allowPyImport, importConfig->allowBare);
        std::string rv = "libsplw.get_safe(scope_vars, '" + ids.at(0)->GetContents() + "', " + boolToString(importConfig->allowImport) + ", " + boolToString(importConfig->allowPyImport) + ", " + boolToString(importConfig->allowInstall) + ", " + boolToString(importConfig->allowBare) + ")";

        if (accessPresent) {
            for (int i = 1; i < ids.size() - 1; i++) {
                rv = "getattr(" + rv + ", \"" + ids[i]->GetContents() + "\")";
            }

            cachedGetattrMinusOne = rv;

            cachedGetattr = "getattr(" + rv + ", \"" + ids.at(ids.size() - 1)->GetContents() + "\")";
        }
        else
            cachedGetattr = rv;

        return minus_one ? cachedGetattrMinusOne : cachedGetattr;
    }

        std::string IdentNode::GenerateGetattrTree(ImportConfig *importConfig, bool minus_one) {
        bool dummy_var = false;
        return GenerateGetattrTree(importConfig, dummy_var, false);
    }

        std::string IdentNode::GetFinalId() {
        return ids.at(ids.size() - 1)->GetContents();
    }
}
```

```
    1: #include "Parser.h"
    2: #include "UtilFunctions.h"
    3:
    4: namespace Spliwaca
    5: {
    6:         template<typename T>
    7:         bool itemInVect(const std::vector<T>& v, T t)
    8:         {
    9:                 for (T e : v)
   10:                 {
   11:                         if (e == t)
   12:                         {
   13:                                 return true;
   14:                         }
   15:                 }
   16:                 return false;
   17:         }
   18:
   19:         std::shared_ptr<Parser> Parser::Create(std::shared_ptr<std::vector<std::shared_ptr<Token>>> toke
ns)
   20:         {
   21:                 return std::shared_ptr<Parser>(new Parser(tokens));
   22:         }
   23:
   24:         std::shared_ptr<EntryPoint> Parser::ConstructAST()
   25:         {
   26:                 std::shared_ptr<EntryPoint> ep = std::make_shared<EntryPoint>();
   27:                 //m_MainScope = std::make_shared<Scope>("Main", 0, ScopeType::Main);
   28:                 //m_CurrentScope = m_MainScope;
   29:                 //m_ScopeStack.push_back(m_MainScope);
   30:
   31:                 ep->require = ConstructRequire();
   32:
   33:                 // Consume newline after require
   34:                 if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Newline && ep->require)
   35:                 {
   36:                         RegisterSyntaxError(SyntaxErrorType::expNewline, m_Tokens->at(m_TokenIndex));
   37:                 }
   38:                 else if (ep->require)
   39:                         IncIndex();
   40:
   41:                 //Begin constructing statements
   42:                 ep->statements = ConstructStatements();
   43:
   44:                 if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::eof)
   45:                 {
   46:                         RegisterSyntaxError(SyntaxErrorType::expNewline, m_Tokens->at(m_TokenIndex));
   47:                         return ep;
   48:                 }
   49:                 else
   50:                         return ep;
   51:         }
   52:
   53:         std::shared_ptr<RequireNode> Parser::ConstructRequire()
   54:         {
   55:                 if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Require)
   56:                 {
   57:                         std::shared_ptr<RequireNode> node = std::make_shared<RequireNode>();
   58:                         IncIndex();
   59:                         node->requireType = ConstructIdentNode();
   60:                         return node;
   61:                 }
   62:                 return nullptr;
   63:         }
   64:
   65:         std::shared_ptr<Statements> Parser::ConstructStatements()
   66:         {
   67:                 std::shared_ptr<Statements> statements = std::make_shared<Statements>();
   68:                 while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::eof)
   69:                 {
   70:                         //Attempt to consume newline
   71:                         while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
   72:                         {
   73:                                 IncIndex();
   74:                         }
   75:
   76:                         TokenType currTokType = m_Tokens->at(m_TokenIndex)->GetType();
   77:                         if (currTokType == TokenType::eof)
   78:                         {
   79:                                 //If we have reached the end of the file, return
   80:                                 break;
   81:                         }
   82:                         else if (currTokType == TokenType::End)
   83:                         {
   84:                                 //Check whether this END matches the type of statement block we are in.
If not, register a syntax error complaining before any
```

```
   85:                                  //other error.
   86:                                  TokenType nextTokType = m_Tokens->at(m_TokenIndex+1)->GetType();
   87:                                  return statements;
   88:                                  //ScopeType currentScopeType = m_CurrentScope->GetType();
   89:                                  if (nextTokType == TokenType::Function)// && currentScopeType != ScopeTy
pe::Function)
   90:                                  {
   91:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndFunc, m_Tokens->at(
m_TokenIndex));
   92:                                  }
   93:                                  else if (nextTokType == TokenType::Procedure)// && currentScopeType != S
copeType::Procedure)
   94:                                  {
   95:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndProc, m_Tokens->at(
m_TokenIndex));
   96:                                  }
   97:                                  else if (nextTokType == TokenType::If)// && currentScopeType != ScopeTyp
e::If)
   98:                                  {
   99:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndIf, m_Tokens->at(m_
TokenIndex));
  100:                                  }
  101:                                  else if (nextTokType == TokenType::For)// && currentScopeType != ScopeTy
pe::For)
  102:                                  {
  103:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndFor, m_Tokens->at(m
_TokenIndex));
  104:                                  }
  105:                                  else if (nextTokType == TokenType::While)// && currentScopeType != Scope
Type::While)
  106:                                  {
  107:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndWhile, m_Tokens->at
(m_TokenIndex));
  108:                                  }
  109:                                  else if (nextTokType == TokenType::Struct)// && currentScopeType != Scop
eType::Struct)
  110:                                  {
  111:                                          RegisterSyntaxError(SyntaxErrorType::unexpEndStruct, m_Tokens->a
t(m_TokenIndex));
  112:                                  }
  113:                                  else
  114:                                  {
  115:                                          return statements;
  116:                                  }
  117:                          }
  118:                          else if (currTokType == TokenType::Else)
  119:                          {
  120:                                  TokenType nextTokType = m_Tokens->at(m_TokenIndex + 1)->GetType();
  121:                                  return statements;
  122:                                  //ScopeType currentScopeType = m_CurrentScope->GetType();
  123:                                  if (nextTokType == TokenType::If)// && currentScopeType != ScopeType::If
)
  124:                                  {
  125:                                          RegisterSyntaxError(SyntaxErrorType::unexpElseIf, m_Tokens->at(m
_TokenIndex));
  126:                                  }
  127:                                  else
  128:                                  {
  129:                                          return statements;
  130:                                  }
  131:                          }
  132:
  133:                          //IncIndex();
  134:                          //Attempt to construct statement
  135:                          std::shared_ptr<Statement> s = ConstructStatement();
  136:                          if (s != nullptr)
  137:                          {
  138:                                  statements->statements.push_back(s);
  139:                          }
  140:                          else
  141:                          {
  142:                                  //If we didn't get a statement back, then there was an error and we are
finished,
  143:                                  //as we do not know what is supposed to happen here.
  144:                                  break;
  145:                          }
  146:                  }
  147:                  return statements;
  148:          }
  149:
  150:          std::shared_ptr<Statement> Parser::ConstructStatement()
  151:          {
  152:                  std::shared_ptr<Statement> s = std::make_shared<Statement>();
  153:                  s->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
  154:                  switch (m_Tokens->at(m_TokenIndex)->GetType())
  155:                  {
```

```
  156:                 case TokenType::If: s->ifNode = ConstructIf(); s->statementType = 0; break;
  157:                 case TokenType::Set: s->setNode = ConstructSet(); s->statementType = 1; break;
  158:                 case TokenType::Input: s->inputNode = ConstructInput(); s->statementType = 2; break;
  159:                 case TokenType::Output: s->outputNode = ConstructOutput(); s->statementType = 3; break;
  160:                 case TokenType::Increment: s->incNode = ConstructIncrement(); s->statementType = 4; brea
k;
  161:                 case TokenType::Decrement: s->decNode = ConstructDecrement(); s->statementType = 5; brea
k;
  162:                 case TokenType::For: s->forNode = ConstructFor(); s->statementType = 6; break;
  163:                 case TokenType::While: s->whileNode = ConstructWhile(); s->statementType = 7; break;
  164:                 case TokenType::Quit: s->quitNode = ConstructQuit(); s->statementType = 8; break;
  165:                 case TokenType::Call: s->callNode = ConstructCall(); s->statementType = 9; break;
  166:                 case TokenType::Function: s->funcNode = ConstructFunction(); s->statementType = 10; brea
k;
  167:                 case TokenType::Procedure: s->procNode = ConstructProcedure(); s->statementType = 11; br
eak;
  168:                 case TokenType::Struct: s->structNode = ConstructStruct(); s->statementType = 12; break;
  169:                 case TokenType::Return: s->returnNode = ConstructReturn(); s->statementType = 13; break;
  170:                 case TokenType::Import: s->importNode = ConstructImport(); s->statementType = 14; break;
  171:                 case TokenType::NoImport: s->statementType = 15; IncIndex(); break;
  172:                 case TokenType::NoInstall: s->statementType = 16; IncIndex(); break;
  173:                 case TokenType::NoBare: s->statementType = 17; IncIndex(); break;
  174:                 default:
  175:                         RegisterSyntaxError(SyntaxErrorType::expStatement, m_Tokens->at(m_TokenIndex));
  176:                         return nullptr;
  177:                 }
  178:                 return s;
  179:         }
  180:
  181:         std::shared_ptr<IfNode> Parser::ConstructIf()
  182:         {
  183:                 std::shared_ptr<IfNode> node = std::make_shared<IfNode>();
  184:                 node->lineNumbers.push_back(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  185:                 IncIndex();
  186:
  187:                 node->conditions.push_back(ConstructList());
  188:                 if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Do)
  189:                 {
  190:                         //There must be a "DO"
  191:                         RegisterSyntaxError(SyntaxErrorType::expDo, m_Tokens->at(m_TokenIndex));
  192:                 }
  193:                 else
  194:                         IncIndex();
  195:
  196:                 //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("IF_line_"+std::to_string(m_Tokens-
>at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::If));
  197:                 //m_CurrentScope = m_ScopeStack.back();
  198:
  199:                 node->bodies.push_back(ConstructStatements());
  200:
  201:                 //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  202:                 //m_ScopeStack.pop_back();
  203:                 //m_CurrentScope = m_ScopeStack.back();
  204:
  205:                 while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Else)
  206:                 {
  207:                         node->lineNumbers.push_back(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  208:                         if (node->elsePresent)
  209:                         {
  210:                                 //We cannot have more than one else
  211:                                 RegisterSyntaxError(SyntaxErrorType::tooManyElse, m_Tokens->at(m_TokenIn
dex));
  212:                         }
  213:                         IncIndex();
  214:                         if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::If)
  215:                         {
  216:                                 IncIndex();
  217:                                 node->conditions.push_back(ConstructList());
  218:                                 if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Do)
  219:                                 {
  220:                                         //There must be a "DO"
  221:                                         RegisterSyntaxError(SyntaxErrorType::expDo, m_Tokens->at(m_Token
Index));
  222:                                 }
  223:                         }
  224:                         else if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Do)
  225:                         {
  226:                                 //There must be a "DO"
  227:                                 RegisterSyntaxError(SyntaxErrorType::expDo, m_Tokens->at(m_TokenIndex));
  228:                         }
  229:                         else
  230:                         {
  231:                                 node->elsePresent = true;
  232:                         }
  233:                         IncIndex();
  234:                         //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("IF_line_" + std::to_string
```

```
(m_Tokens->at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::If));
  235:                          //m_CurrentScope = m_ScopeStack.back();
  236:
  237:                          node->bodies.push_back(ConstructStatements());
  238:
  239:                          //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  240:                          //m_ScopeStack.pop_back();
  241:                          //m_CurrentScope = m_ScopeStack.back();
  242:                  }
  243:                  if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex+(uint64_t)1)->GetType() == TokenType::If)
  244:                  {
  245:                          IncIndex(); IncIndex();
  246:                  }
  247:                  else
  248:                  {
  249:                          //Must have an end if - something is wrong here
  250:                          RegisterSyntaxError(SyntaxErrorType::expEndIf, m_Tokens->at(m_TokenIndex));
  251:                  }
  252:                  return node;
  253:          }
  254:
  255:          std::shared_ptr<SetNode> Parser::ConstructSet()
  256:          {
  257:                  std::shared_ptr<SetNode> node = std::make_shared<SetNode>();
  258:                  IncIndex();
  259:                  node->id = ConstructIdentNode();
  260:
  261:                  if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::To)
  262:                  {
  263:                          RegisterSyntaxError(SyntaxErrorType::expTo, m_Tokens->at(m_TokenIndex));
  264:                  }
  265:                  else
  266:                          IncIndex();
  267:
  268:                  node->list = ConstructList();
  269:                  //VarType varType;
  270:                  /*switch (node->expr->exprType)
  271:                  {
  272:                  case 1:
  273:                  {
  274:                          if (node->expr->listNode->Items.size() > 1 || node->expr->listNode->Items.at(0)-
>hasRight)
  275:                          {
  276:                                  varType = VarType::List;
  277:                          }
  278:                          else
  279:                          {
  280:                                  if (node->expr->listNode->Items.at(0)->left->right != nullptr)
  281:                                  {
  282:                                          varType = VarType::Bool;
  283:                                  }
  284:                                  else
  285:                                  {
  286:                                          auto atom = node->expr->listNode->Items.at(0)->left->left->left-
>left->left->left->right;
  287:                                          while (atom->expression)
  288:                                          {
  289:                                                  atom = atom->expression->listNode->Items.at(0)->left->le
ft->left->left->left->left->right;
  290:                                          }
  291:                                          if (atom->ident != nullptr)
  292:                                          {
  293:                                                  varType = m_CurrentScope->FindIdent(atom->ident)->GetSym
bolType();
  294:                                          }
  295:                                          else if (atom->token->GetType() == TokenType::Int)
  296:                                          {
  297:
  298:                                          }
  299:                                  }
  300:                          }
  301:                  }
  302:                  }*/
  303:
  304:                  //m_CurrentScope->AddEntry(node->id->GetContents(), node->id->GetLineNumber(), node->exp
r->GetExprReturnType());
  305:                  return node;
  306:          }
  307:
  308:          std::shared_ptr<InputNode> Parser::ConstructInput()
  309:          {
  310:                  std::shared_ptr<InputNode> node = std::make_shared<InputNode>();
  311:
  312:                  IncIndex();
  313:                  auto type = m_Tokens->at(m_TokenIndex)->GetType();
```

```
  314:                if (type == TokenType::PositiveTypeMod || type == TokenType::NegativeTypeMod || type ==
TokenType::NonZeroTypeMod)
  315:                {
  316:                        node->signSpec = m_Tokens->at(m_TokenIndex);
  317:                        IncIndex();
  318:                        if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Type) {
  319:                                RegisterSyntaxError(SyntaxErrorType::expTypeMod, m_Tokens->at(m_TokenInd
ex));
  320:                        } else {
  321:                                node->type = ConstructTypeNode();
  322:                        }
  323:                }
  324:                else if (type == TokenType::Type)
  325:                {
  326:                        node->type = ConstructTypeNode();
  327:                }
  328:                else
  329:                {
  330:                        RegisterSyntaxError(SyntaxErrorType::expTypeMod, m_Tokens->at(m_TokenIndex));
  331:                }
  332:                //IncIndex();
  333:
  334:                node->id = ConstructIdentNode();
  335:                //m_CurrentScope->AddEntry(node->id->GetContents(), node->id->GetLineNumber());
  336:
  337:                return node;
  338:        }
  339:
  340:        std::shared_ptr<OutputNode> Parser::ConstructOutput()
  341:        {
  342:                std::shared_ptr<OutputNode> node = std::make_shared<OutputNode>();
  343:                IncIndex();
  344:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Raw)
  345:                {
  346:                        RegisterSyntaxError(SyntaxErrorType::expRaw, m_Tokens->at(m_TokenIndex));
  347:                }
  348:                else
  349:                {
  350:                        node->raw = m_Tokens->at(m_TokenIndex);
  351:                        IncIndex();
  352:                }
  353:                return node;
  354:        }
  355:
  356:        std::shared_ptr<IncNode> Parser::ConstructIncrement()
  357:        {
  358:                std::shared_ptr<IncNode> node = std::make_shared<IncNode>();
  359:                IncIndex();
  360:                node->id = ConstructIdentNode();
  361:                return node;
  362:        }
  363:
  364:        std::shared_ptr<DecNode> Parser::ConstructDecrement()
  365:        {
  366:                std::shared_ptr<DecNode> node = std::make_shared<DecNode>();
  367:                IncIndex();
  368:                node->id = ConstructIdentNode();
  369:                return node;
  370:        }
  371:
  372:        std::shared_ptr<ForNode> Parser::ConstructFor()
  373:        {
  374:                std::shared_ptr<ForNode> node = std::make_shared<ForNode>();
  375:                node->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
  376:                IncIndex();
  377:
  378:                //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("FOR_line_" + std::to_string(m_Toke
ns->at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::For));
  379:                //m_CurrentScope = m_ScopeStack.back();
  380:                node->id = ConstructIdentNode();
  381:
  382:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Of)
  383:                {
  384:                        RegisterSyntaxError(SyntaxErrorType::expOf, m_Tokens->at(m_TokenIndex));
  385:                }
  386:                else
  387:                        IncIndex();
  388:
  389:                node->iterableExpr = ConstructList();
  390:
  391:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Do)
  392:                {
  393:                        RegisterSyntaxError(SyntaxErrorType::expDo, m_Tokens->at(m_TokenIndex));
  394:                }
  395:                IncIndex();
  396:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Newline)
```

```
  397:                {
  398:                        RegisterSyntaxError(SyntaxErrorType::expNewline, m_Tokens->at(m_TokenIndex));
  399:                }
  400:
  401:                node->body = ConstructStatements();
  402:
  403:                if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::For)
  404:                {
  405:                        //CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  406:                        IncIndex(); IncIndex();
  407:                }
  408:                else
  409:                {
  410:                        //CurrentScope->CloseScope(0);
  411:                        RegisterSyntaxError(SyntaxErrorType::expEndFor, m_Tokens->at(m_TokenIndex));
  412:                }
  413:
  414:                //ScopeStack.pop_back();
  415:                //CurrentScope = m_ScopeStack.back();
  416:                return node;
  417:        }
  418:
  419:        std::shared_ptr<WhileNode> Parser::ConstructWhile()
  420:        {
  421:                std::shared_ptr<WhileNode> node = std::make_shared<WhileNode>();
  422:                node->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
  423:                IncIndex();
  424:
  425:                //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("WHILE_line_" + std::to_string(m_To
kens->at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::While));
  426:                //m_CurrentScope = m_ScopeStack.back();
  427:
  428:                node->condition = ConstructBinOpNode();
  429:
  430:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Do)
  431:                {
  432:                        RegisterSyntaxError(SyntaxErrorType::expDo, m_Tokens->at(m_TokenIndex));
  433:                }
  434:                IncIndex();
  435:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Newline)
  436:                {
  437:                        RegisterSyntaxError(SyntaxErrorType::expNewline, m_Tokens->at(m_TokenIndex));
  438:                }
  439:
  440:                node->body = ConstructStatements();
  441:
  442:                if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::While)
  443:                {
  444:                        //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
  445:                        IncIndex(); IncIndex();
  446:                }
  447:                else
  448:                {
  449:                        //m_CurrentScope->CloseScope(0);
  450:                        RegisterSyntaxError(SyntaxErrorType::expEndWhile, m_Tokens->at(m_TokenIndex));
  451:                }
  452:
  453:                //m_ScopeStack.pop_back();
  454:                //m_CurrentScope = m_ScopeStack.back();
  455:                return node;
  456:        }
  457:
  458:        std::shared_ptr<QuitNode> Parser::ConstructQuit()
  459:        {
  460:                std::shared_ptr<QuitNode> node = std::make_shared<QuitNode>();
  461:                IncIndex();
  462:                if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Newline)
  463:                        node->returnVal = ConstructAtom(true);
  464:                return node;
  465:        }
  466:
  467:        std::shared_ptr<CallNode> Parser::ConstructCall()
  468:        {
  469:                std::shared_ptr<CallNode> node = std::make_shared<CallNode>();
  470:                IncIndex();
  471:                node->function = ConstructExpr();
  472:
  473:                if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::With)
  474:                {
  475:                        IncIndex();
  476:
  477:                        node->args.push_back(ConstructExpr());
  478:
  479:                        while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Comma)
```

```
480:                            {
481:                                    IncIndex();
482:
483:                                    node->args.push_back(ConstructExpr());
484:                            }
485:                    }
486:                    return node;
487:            }
488:
489:            std::shared_ptr<ImportNode> Parser::ConstructImport()
490:            {
491:                    std::shared_ptr<ImportNode> node = std::make_shared<ImportNode>();
492:                    IncIndex();
493:
494:                    node->id = ConstructIdentNode();
495:
496:                    return node;
497:            }
498:
499:            std::shared_ptr<FuncNode> Parser::ConstructFunction()
500:            {
501:                    std::shared_ptr<FuncNode> node = std::make_shared<FuncNode>();
502:                    node->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
503:                    IncIndex();
504:
505:                    node->id = ConstructIdentNode();
506:
507:                    //m_CurrentScope->AddEntry(node->id->GetContents(), node->id->GetLineNumber());
508:                    //m_ScopeStack.push_back(m_CurrentScope->AddSubScope(node->id->GetContents(), node->id->
GetLineNumber(), ScopeType::Function));
509:                    //m_CurrentScope = m_ScopeStack.back();
510:
511:                    if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Takes)
512:                    {
513:                            IncIndex();
514:                            node->argTypes.push_back(ConstructTypeNode());
515:                            node->argNames.push_back(ConstructIdentNode());
516:                            //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->argNames.
back()->GetLineNumber());
517:
518:                            while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::ReturnType)
519:                            {
520:                                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Comma)
521:                                    {
522:                                            RegisterSyntaxError(SyntaxErrorType::expComma, m_Tokens->at(m_To
kenIndex));
523:                                    }
524:                                    else
525:                                            IncIndex();
526:                                    node->argTypes.push_back(ConstructTypeNode());
527:                                    node->argNames.push_back(ConstructIdentNode());
528:                                    //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->a
rgNames.back()->GetLineNumber());
529:                            }
530:                    }
531:
532:                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::ReturnType)
533:                    {
534:                            RegisterSyntaxError(SyntaxErrorType::expReturns, m_Tokens->at(m_TokenIndex));
535:                    }
536:                    else
537:                            IncIndex();
538:
539:                    node->returnType = ConstructTypeNode();
540:
541:                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
542:                    {
543:                            RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
544:                    }
545:                    else
546:                            IncIndex();
547:
548:                    node->body = ConstructStatements();
549:
550:                    if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::Function)
551:                    {
552:                            //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
553:                            IncIndex(); IncIndex();
554:                    }
555:                    else
556:                    {
557:                            RegisterSyntaxError(SyntaxErrorType::expEndFunc, m_Tokens->at(m_TokenIndex));
558:                            //m_CurrentScope->CloseScope(0);
559:                    }
560:
```

```
561:                    //m_ScopeStack.pop_back();
562:                    //m_CurrentScope = m_ScopeStack.back();
563:
564:                    return node;
565:            }
566:
567:            std::shared_ptr<ProcNode> Parser::ConstructProcedure()
568:            {
569:                    std::shared_ptr<ProcNode> node = std::make_shared<ProcNode>();
570:                    node->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
571:                    IncIndex();
572:
573:                    node->id = ConstructIdentNode();
574:
575:                    //m_CurrentScope->AddEntry(node->id->GetContents(), node->id->GetLineNumber());
576:                    //m_ScopeStack.push_back(m_CurrentScope->AddSubScope(node->id->GetContents(), node->id->
GetLineNumber(), ScopeType::Procedure));
577:                    //m_CurrentScope = m_ScopeStack.back();
578:
579:                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
580:                    {
581:                            if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Takes)
582:                            {
583:                                    RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
584:                            }
585:                            else
586:                                    IncIndex();
587:                            node->argTypes.push_back(ConstructTypeNode());
588:                            node->argNames.push_back(ConstructIdentNode());
589:                            //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->argNames.
back()->GetLineNumber());
590:
591:                            while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
592:                            {
593:                                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Comma)
594:                                    {
595:                                            RegisterSyntaxError(SyntaxErrorType::expComma, m_Tokens->at(m_To
kenIndex));
596:                                    }
597:                                    else
598:                                            IncIndex();
599:                                    node->argTypes.push_back(ConstructTypeNode());
600:                                    node->argNames.push_back(ConstructIdentNode());
601:                                    //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->a
rgNames.back()->GetLineNumber());
602:                            }
603:                    }
604:
605:                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
606:                    {
607:                            RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
608:                    }
609:                    else
610:                            IncIndex();
611:
612:                    node->body = ConstructStatements();
613:
614:                    if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::Procedure)
615:                    {
616:                            //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
617:                            IncIndex(); IncIndex();
618:                    }
619:                    else
620:                    {
621:                            //m_CurrentScope->CloseScope(0);
622:                            RegisterSyntaxError(SyntaxErrorType::expEndProc, m_Tokens->at(m_TokenIndex));
623:                    }
624:
625:                    //m_ScopeStack.pop_back();
626:                    //m_CurrentScope = m_ScopeStack.back();
627:                    return node;
628:            }
629:
630:            std::shared_ptr<StructNode> Parser::ConstructStruct()
631:            {
632:                    std::shared_ptr<StructNode> node = std::make_shared<StructNode>();
633:                    node->lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
634:                    IncIndex();
635:
636:                    node->id = ConstructIdentNode();
637:
638:                    //m_CurrentScope->AddEntry(node->id->GetContents(), node->id->GetLineNumber());
639:                    //m_ScopeStack.push_back(m_CurrentScope->AddSubScope(node->id->GetContents(), m_Tokens->
at(m_TokenIndex)->GetLineNumber(), ScopeType::If));
640:                    //m_CurrentScope = m_ScopeStack.back();
```

```
641:
642:
643:
644:                    if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
645:                    {
646:                            RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
647:                    }
648:                    else
649:                            IncIndex();
650:
651:                    node->types.push_back(ConstructTypeNode());
652:
653:                    node->names.push_back(ConstructIdentNode());
654:
655:                    if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
656:                    {
657:                            IncIndex();
658:                    }
659:
660:                    while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::End)
661:                    {
662:                            if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Comma)
663:                            {
664:                                    RegisterSyntaxError(SyntaxErrorType::expComma, m_Tokens->at(m_TokenIndex
));
665:                            }
666:                            else
667:                                    IncIndex();
668:
669:                            if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
670:                            {
671:                                    IncIndex();
672:                            }
673:
674:                            node->types.push_back(ConstructTypeNode());
675:
676:                            node->names.push_back(ConstructIdentNode());
677:
678:                            if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
679:                            {
680:                                    IncIndex();
681:                            }
682:                    }
683:
684:                    if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::Struct)
685:                    {
686:                            //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
687:                            IncIndex(); IncIndex();
688:                    }
689:                    else
690:                    {
691:                            //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
692:                            RegisterSyntaxError(SyntaxErrorType::expEndStruct, m_Tokens->at(m_TokenIndex));
693:                    }
694:
695:                    //m_ScopeStack.pop_back();
696:                    //m_CurrentScope = m_ScopeStack.back();
697:                    return node;
698:            }
699:
700:            std::shared_ptr<Expr> Parser::ConstructExpr()
701:            {
702:                    std::shared_ptr<Expr> node = std::make_shared<Expr>();
703:                    switch (m_Tokens->at(m_TokenIndex)->GetType())
704:                    {
705:                    case TokenType::Create:
706:                            node->createNode = ConstructCreate();
707:                            node->exprType = 2; break;
708:                    case TokenType::Cast:
709:                            node->castNode = ConstructCast();
710:                            node->exprType = 3; break;
711:                    case TokenType::Call:
712:                            node->callNode = ConstructCall();
713:                            node->exprType = 4; break;
714:                    case TokenType::AnonFunc:
715:                            node->anonfNode = ConstructAnonFunc();
716:                            node->exprType = 5; break;
717:                    case TokenType::AnonProc:
718:                            node->anonpNode = ConstructAnonProc();
719:                            node->exprType = 6; break;
720:                    default:
721:                            node->binOpNode = ConstructBinOpNode();
722:                            node->exprType = 1; break;
723:                    }
724:                    return node;
```

```
725:         }
726:
727:         std::shared_ptr<ListNode> Parser::ConstructList()
728:         {
729:                 std::shared_ptr<ListNode> node = std::make_shared<ListNode>();
730:                 uint32_t lineNumber = m_Tokens->at(m_TokenIndex)->GetLineNumber();
731:
732:                 node->Items.push_back(ConstructDictEntry());
733:                 while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Comma)
734:                 {
735:                         IncIndex();
736:                         if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
737:                                 IncIndex();
738:                         node->Items.push_back(ConstructDictEntry());
739:                         if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Newline)
740:                                 IncIndex();
741:                 }
742:
743:                 int dictPairs = -1;
744:                 for (auto dictNode : node->Items)
745:                 {
746:                         if (dictNode->hasRight && dictPairs == 0)
747:                                 RegisterSyntaxError(SyntaxErrorType::inconsistentDict, lineNumber, 0, 10
, TokenType::None);
748:                         else if (dictNode->hasRight)
749:                                 dictPairs = 1;
750:                         else if (dictPairs == 1)
751:                                 RegisterSyntaxError(SyntaxErrorType::inconsistentDict, lineNumber, 0, 10
, TokenType::None);
752:                         else
753:                                 dictPairs = 0;
754:                 }
755:
756:                 return node;
757:         }
758:
759:         std::shared_ptr<DictEntryNode> Parser::ConstructDictEntry()
760:         {
761:                 std::shared_ptr<DictEntryNode> node = std::make_shared<DictEntryNode>();
762:                 node->left = ConstructExpr();
763:                 if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::DictEquator)
764:                 {
765:                         IncIndex();
766:                         node->right = ConstructExpr();
767:                         node->hasRight = true;
768:                 }
769:                 else
770:                         node->hasRight = false;
771:
772:                 return node;
773:         }
774:
775:         /*
776:         std::shared_ptr<BoolExprNode> Parser::ConstructBooleanExpr()
777:         {
778:                 std::shared_ptr<BoolExprNode> node = std::make_shared<BoolExprNode>();
779:                 if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Not)
780:                 {
781:                         node->exprType = 1;
782:                         IncIndex();
783:                         node->right = ConstructBooleanExpr();
784:                 }
785:                 else
786:                 {
787:
788:                         node->left = ConstructAddExpr();
789:                         TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
790:                         if (type == TokenType::NotEqual || type == TokenType::Equal || type == TokenType
::LessThan || type == TokenType::GreaterThan || type == TokenType::LessThanEqual || type == TokenType::GreaterT
hanEqual)
791:                         {
792:                                 node->exprType = 2;
793:                                 node->opToken = m_Tokens->at(m_TokenIndex);
794:                                 IncIndex();
795:                                 node->right = ConstructBooleanExpr();
796:                         }
797:                         else
798:                         {
799:                                 node->exprType = 3;
800:                         }
801:                 }
802:                 return node;
803:         }
804:
805:         std::shared_ptr<AddExprNode> Parser::ConstructAddExpr()
806:         {
```

```
  807:                 std::shared_ptr<AddExprNode> node = std::make_shared<AddExprNode>();
  808:                 node->left = ConstructMulExpr();
  809:                 TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
  810:                 if (type == TokenType::Plus || type == TokenType::Minus)
  811:                 {
  812:                         node->opToken = m_Tokens->at(m_TokenIndex);
  813:                         IncIndex();
  814:                         node->right = ConstructAddExpr();
  815:                 }
  816:                 return node;
  817:         }
  818:
  819:         std::shared_ptr<MulExprNode> Parser::ConstructMulExpr()
  820:         {
  821:                 std::shared_ptr<MulExprNode> node = std::make_shared<MulExprNode>();
  822:                 node->left = ConstructDivModExpr();
  823:                 TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
  824:                 if (type == TokenType::Multiply || type == TokenType::Divide)
  825:                 {
  826:                         node->opToken = m_Tokens->at(m_TokenIndex);
  827:                         IncIndex();
  828:                         node->right = ConstructMulExpr();
  829:                 }
  830:                 return node;
  831:         }
  832:
  833:         std::shared_ptr<DivModExprNode> Parser::ConstructDivModExpr()
  834:         {
  835:                 std::shared_ptr<DivModExprNode> node = std::make_shared<DivModExprNode>();
  836:                 node->left = ConstructPower();
  837:                 TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
  838:                 if (type == TokenType::Intdiv || type == TokenType::Modulo)
  839:                 {
  840:                         node->opToken = m_Tokens->at(m_TokenIndex);
  841:                         IncIndex();
  842:                         node->right = ConstructDivModExpr();
  843:                 }
  844:                 return node;
  845:         }
  846:
  847:         std::shared_ptr<PowerNode> Parser::ConstructPower()
  848:         {
  849:                 std::shared_ptr<PowerNode> node = std::make_shared<PowerNode>();
  850:                 node->left = ConstructFactor();
  851:                 TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
  852:                 if (type == TokenType::Power)
  853:                 {
  854:                         node->opToken = m_Tokens->at(m_TokenIndex);
  855:                         IncIndex();
  856:                         node->right = ConstructPower();
  857:                 }
  858:                 return node;
  859:         }
  860:         */
  861:
  862:         std::shared_ptr<BinOpNode> Parser::ConstructBinOpNode()
  863:         {
  864:                 std::shared_ptr<BinOpNode> node = std::make_shared<BinOpNode>();
  865:                 node->left = ConstructFactor();
  866:                 std::vector<TokenType> acceptedTypes = { TokenType::And, TokenType::Or, TokenType::NotEq
ual, TokenType::Equal, TokenType::GreaterThan,
  867:                         TokenType::GreaterThanEqual, TokenType::LessThan , TokenType::LessThanEqual, Tok
enType::Plus, TokenType::Minus, TokenType::Modulo,
  868:                         TokenType::Multiply, TokenType::Divide, TokenType::Intdiv, TokenType::Power, Tok
enType::BitwiseAnd, TokenType::BitwiseOr};
  869:
  870:                 if (itemInVect(acceptedTypes, m_Tokens->at(m_TokenIndex)->GetType()))
  871:                 {
  872:                         node->opToken = m_Tokens->at(m_TokenIndex);
  873:                         IncIndex();
  874:                         node->right = ConstructBinOpNode();
  875:                 }
  876:                 return node;
  877:         }
  878:
  879:         std::shared_ptr<FactorNode> Parser::ConstructFactor()
  880:         {
  881:                 std::shared_ptr<FactorNode> node = std::make_shared<FactorNode>();
  882:                 std::vector<TokenType> acceptedTypes = { TokenType::Plus, TokenType::Minus, TokenType::N
ot };
  883:                 if (itemInVect(acceptedTypes, m_Tokens->at(m_TokenIndex)->GetType()))
  884:                 {
  885:                         node->opToken = m_Tokens->at(m_TokenIndex);
  886:                         IncIndex();
  887:                         node->right = ConstructAtom();
  888:                         node->opTokenPresent = true;
```

```
  889:                }
  890:                else
  891:                {
  892:                        node->right = ConstructAtom();
  893:                        node->opTokenPresent = false;
  894:                }
  895:                return node;
  896:        }
  897:
  898:        std::shared_ptr<AtomNode> Parser::ConstructAtom(bool quit)
  899:        {
  900:                std::shared_ptr<AtomNode> node = std::make_shared<AtomNode>();
  901:                if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::LParen)
  902:                {
  903:                        IncIndex();
  904:                        node->list = ConstructList();
  905:                        if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::RParen)
  906:                        {
  907:                                RegisterSyntaxError(SyntaxErrorType::expRParen, m_Tokens->at(m_TokenInde
x));
  908:                        }
  909:                        else
  910:                                IncIndex();
  911:                        node->type = 2;
  912:                }
  913:                else
  914:                {
  915:                        std::vector<TokenType> acceptedAtomTokenTypes = { TokenType::String, TokenType::
Raw, TokenType::Int, TokenType::Float, TokenType::Complex, TokenType::Identifier, TokenType::None };
  916:                        TokenType type = m_Tokens->at(m_TokenIndex)->GetType();
  917:                        if (!itemInVect(acceptedAtomTokenTypes, type) && !(((m_Tokens->at(m_TokenIndex)-
>GetContents()[0] & ~0x20) >= 'A') && ((m_Tokens->at(m_TokenIndex)->GetContents()[0] & ~0x20) <= 'Z')))
  918:                        {
  919:                                //If it doesn't start with a valid ident, then it isn't an identifier, a
nd if it isn't any of the others, then it must be an error
  920:                                if (quit)
  921:                                        return nullptr;
  922:                                RegisterSyntaxError(SyntaxErrorType::expAtom, m_Tokens->at(m_TokenIndex)
);
  923:                                node->type = 0;
  924:                                IncIndex();
  925:                        }
  926:                        else if (type == TokenType::Identifier || ((m_Tokens->at(m_TokenIndex)->GetConte
nts()[0] & ~0x20) >= 'A' && (m_Tokens->at(m_TokenIndex)->GetContents()[0] & ~0x20) <= 'Z' && type != TokenType:
:String && type != TokenType::Raw))
  927:                        {
  928:                                //std::shared_ptr<IdentNode> ident = ConstructIdentNode();
  929:                                node->ident = ConstructIdentNode();
  930:                                node->type = 3;
  931:                                //@IMPORTANT How are global variables managed? Are all variables declare
d in the main scope treated as globals, or do
  932:                                //           they need to be specially declared and placed in a global s
cope? main scope = global scope, scope state -> global variables
  933:                                // it is NOT an error to reference a variable from a previous scope befo
re defining it in the current scope - different to python behaviour
  934:                                // w/ functions
  935:                                /*if (m_CurrentScope->FindIdent(ident) || m_MainScope->FindIdent(ident))
  936:                                {
  937:                                        node->ident = ident;
  938:                                        node->type = 3;
  939:                                }
  940:                                else
  941:                                {
  942:                                        //RegisterMissingVariable(ident->GetLineNumber(), ident->GetColu
mnNumber());
  943:                                        node->type = 0;
  944:                                }*/
  945:                        }
  946:                        else if (itemInVect(acceptedAtomTokenTypes, type))
  947:                        {
  948:                                node->token = m_Tokens->at(m_TokenIndex);
  949:                                node->type = 1;
  950:                                IncIndex();
  951:                        }
  952:                        else {
  953:                                SPLW_CRITICAL("We shouldn't be here!");
  954:                        }
  955:                }
  956:
  957:                if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::LSquareParen)
  958:                {
  959:                        node->listAccess = ConstructListAccess();
  960:                        node->listAccessPresent = true;
  961:                }
  962:                return node;
  963:        }
```

```
  964:
  965:          std::shared_ptr<ListAccessNode> Parser::ConstructListAccess()
  966:          {
  967:                  std::shared_ptr<ListAccessNode> node = std::make_shared<ListAccessNode>();
  968:                  IncIndex();
  969:
  970:                  node->indices.push_back(ConstructList());
  971:                  if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::RSquareParen)
  972:                          IncIndex();
  973:                  else
  974:                          RegisterSyntaxError(SyntaxErrorType::expRSquareParen, m_Tokens->at(m_TokenIndex)
);
  975:
  976:                  while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::LSquareParen)
  977:                  {
  978:                          node->indices.push_back(ConstructList());
  979:                          if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::RSquareParen)
  980:                                  IncIndex();
  981:                          else
  982:                                  RegisterSyntaxError(SyntaxErrorType::expRSquareParen, m_Tokens->at(m_Tok
enIndex));
  983:                  }
  984:
  985:                  return std::shared_ptr<ListAccessNode>();
  986:          }
  987:
  988:          std::shared_ptr<CreateNode> Parser::ConstructCreate()
  989:          {
  990:                  std::shared_ptr<CreateNode> node = std::make_shared<CreateNode>();
  991:                  IncIndex();
  992:                  node->createType = ConstructTypeNode();
  993:
  994:                  if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::With)
  995:                  {
  996:                          node->args.push_back(ConstructExpr());
  997:                          while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Comma)
  998:                          {
  999:                                  node->args.push_back(ConstructExpr());
 1000:                          }
 1001:                  }
 1002:                  return node;
 1003:          }
 1004:
 1005:          std::shared_ptr<CastNode> Parser::ConstructCast()
 1006:          {
 1007:                  std::shared_ptr<CastNode> node = std::make_shared<CastNode>();
 1008:                  IncIndex();
 1009:                  node->castType = ConstructTypeNode();
 1010:                  node->list = ConstructList();
 1011:                  return node;
 1012:          }
 1013:
 1014:          std::shared_ptr<ReturnNode> Parser::ConstructReturn()
 1015:          {
 1016:                  std::shared_ptr<ReturnNode> node = std::make_shared<ReturnNode>();
 1017:                  IncIndex();
 1018:                  node->list = ConstructList();
 1019:                  return node;
 1020:          }
 1021:
 1022:          std::shared_ptr<AnonfNode> Parser::ConstructAnonFunc()
 1023:          {
 1024:                  std::shared_ptr<AnonfNode> node = std::make_shared<AnonfNode>();
 1025:                  IncIndex();
 1026:
 1027:                  //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("ANONF_line_" + std::to_string(m_To
kens->at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::Anonf));
 1028:                  //m_CurrentScope = m_ScopeStack.back();
 1029:
 1030:                  if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Takes)
 1031:                  {
 1032:                          IncIndex();
 1033:                          node->argTypes.push_back(ConstructTypeNode());
 1034:                          node->argNames.push_back(ConstructIdentNode());
 1035:                          //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->argNames.
back()->GetLineNumber());
 1036:
 1037:                          while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::ReturnType)
 1038:                          {
 1039:                                  if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Comma)
 1040:                                  {
 1041:                                          RegisterSyntaxError(SyntaxErrorType::expComma, m_Tokens->at(m_To
kenIndex));
 1042:                                  }
 1043:                                  else
 1044:                                          IncIndex();
```

```
 1045:                                   node->argTypes.push_back(ConstructTypeNode());
 1046:                                   node->argNames.push_back(ConstructIdentNode());
 1047:                                   //m_CurrentScope->AddEntry(node->argNames.back()->GetContents(), node->a
rgNames.back()->GetLineNumber());
 1048:                           }
 1049:                   }
 1050:
 1051:                   if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::ReturnType)
 1052:                   {
 1053:                           RegisterSyntaxError(SyntaxErrorType::expReturns, m_Tokens->at(m_TokenIndex));
 1054:                   }
 1055:                   else
 1056:                           IncIndex();
 1057:
 1058:                   node->returnType = ConstructTypeNode();
 1059:
 1060:                   if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
 1061:                   {
 1062:                           RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
 1063:                   }
 1064:                   else
 1065:                           IncIndex();
 1066:
 1067:                   node->body = ConstructStatements();
 1068:
 1069:                   if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::Function)
 1070:                   {
 1071:                           //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
 1072:                           IncIndex(); IncIndex();
 1073:                   }
 1074:                   else
 1075:                   {
 1076:                           //m_CurrentScope->CloseScope(0);
 1077:                           RegisterSyntaxError(SyntaxErrorType::expEndFunc, m_Tokens->at(m_TokenIndex));
 1078:                   }
 1079:
 1080:                   //m_ScopeStack.pop_back();
 1081:                   //m_CurrentScope = m_ScopeStack.back();
 1082:                   return node;
 1083:           }
 1084:
 1085:           std::shared_ptr<AnonpNode> Parser::ConstructAnonProc()
 1086:           {
 1087:                   std::shared_ptr<AnonpNode> node = std::make_shared<AnonpNode>();
 1088:                   IncIndex();
 1089:
 1090:                   //m_ScopeStack.push_back(m_CurrentScope->AddSubScope("ANONP_line_" + std::to_string(m_To
kens->at(m_TokenIndex)->GetLineNumber()), m_Tokens->at(m_TokenIndex)->GetLineNumber(), ScopeType::Anonp));
 1091:                   //m_CurrentScope = m_ScopeStack.back();
 1092:
 1093:                   if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
 1094:                   {
 1095:                           if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Takes)
 1096:                           {
 1097:                                   RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
 1098:                           }
 1099:                           else
 1100:                                   IncIndex();
 1101:
 1102:                           node->argTypes.push_back(ConstructTypeNode());
 1103:
 1104:                           node->argNames.push_back(ConstructIdentNode());
 1105:
 1106:                           while (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
 1107:                           {
 1108:                                   if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::Comma)
 1109:                                   {
 1110:                                           RegisterSyntaxError(SyntaxErrorType::expComma, m_Tokens->at(m_To
kenIndex));
 1111:                                   }
 1112:                                   else
 1113:                                           IncIndex();
 1114:
 1115:                                   node->argTypes.push_back(ConstructTypeNode());
 1116:
 1117:                                   node->argNames.push_back(ConstructIdentNode());
 1118:
 1119:                           }
 1120:                   }
 1121:
 1122:                   if (m_Tokens->at(m_TokenIndex)->GetType() != TokenType::As)
 1123:                   {
 1124:                           RegisterSyntaxError(SyntaxErrorType::expAs, m_Tokens->at(m_TokenIndex));
 1125:                   }
 1126:                   else
```

```
 1127:                         IncIndex();
 1128:
 1129:                 node->body = ConstructStatements();
 1130:
 1131:                 if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::End && m_Tokens->at((uint64_t)m_
TokenIndex + (uint64_t)1)->GetType() == TokenType::Procedure)
 1132:                 {
 1133:                         //m_CurrentScope->CloseScope(m_Tokens->at(m_TokenIndex)->GetLineNumber());
 1134:                         IncIndex(); IncIndex();
 1135:                 }
 1136:                 else
 1137:                 {
 1138:                         //m_CurrentScope->CloseScope(0);
 1139:                         RegisterSyntaxError(SyntaxErrorType::expEndFunc, m_Tokens->at(m_TokenIndex));
 1140:                 }
 1141:
 1142:                 //m_ScopeStack.pop_back();
 1143:                 //m_CurrentScope = m_ScopeStack.back();
 1144:                 return node;
 1145:         }
 1146:
 1147:         std::shared_ptr<TypeNode> Parser::ConstructTypeNode()
 1148:         {
 1149:                 std::shared_ptr<TypeNode> node = std::make_shared<TypeNode>();
 1150:                 if (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Type)
 1151:                 {
 1152:                         node->typeToken = m_Tokens->at(m_TokenIndex);
 1153:                         node->type = 2;
 1154:                         IncIndex();
 1155:                 }
 1156:                 else
 1157:                 {
 1158:                         node->ident = ConstructIdentNode();
 1159:                         node->type = 1;
 1160:                 }
 1161:                 return node;
 1162:         }
 1163:
 1164:         std::shared_ptr<IdentNode> Parser::ConstructIdentNode()
 1165:         {
 1166:                 std::shared_ptr<IdentNode> node = std::make_shared<IdentNode>();
 1167:
 1168:                 char first = m_Tokens->at(m_TokenIndex)->GetContents()[0];
 1169:                 if (!(first >= 0x41 && first <= 0x5a) && !(first >= 0x61 && first <= 0x7a) && !(m_Tokens
->at(m_TokenIndex)->GetType() == TokenType::Identifier))
 1170:                 {
 1171:                         RegisterSyntaxError(SyntaxErrorType::expIdent, m_Tokens->at(m_TokenIndex));
 1172:                 }
 1173:                 else
 1174:                 {
 1175:                         node->ids.push_back(m_Tokens->at(m_TokenIndex));
 1176:                         IncIndex();
 1177:                 }
 1178:
 1179:                 while (m_Tokens->at(m_TokenIndex)->GetType() == TokenType::VarAccessOp)
 1180:                 {
 1181:                         IncIndex();
 1182:                         first = m_Tokens->at(m_TokenIndex)->GetContents()[0];
 1183:                         if (!(first >= 0x41 && first <= 0x5a) && !(first >= 0x61 && first <= 0x7a) && !(
m_Tokens->at(m_TokenIndex)->GetType() == TokenType::Identifier)) {
 1184:                                 RegisterSyntaxError(SyntaxErrorType::expIdent, m_Tokens->at(m_TokenIndex
));
 1185:                         } else {
 1186:                                 node->ids.push_back(m_Tokens->at(m_TokenIndex));
 1187:                                 IncIndex();
 1188:                         }
 1189:                         node->accessPresent = true;
 1190:                 }
 1191:                 return node;
 1192:         }
 1193: }
```

```
    1: #include "Generator.h"
    2: #include "UtilFunctions.h"
    3: #include <regex>
    4:
    5: namespace Spliwaca
    6: {
    7:         template<typename T>
    8:         bool itemInVect(const std::vector<T>& v, T t)
    9:         {
   10:                 for (T e : v)
   11:                 {
   12:                         if (e == t)
   13:                         {
   14:                                 return true;
   15:                         }
   16:                 }
   17:                 return false;
   18:         }
   19:
   20: /*      bool charInStr(const std::string& s, char c)
   21:         {
   22:                 if (s.find(c) != std::string::npos)
   23:                         return true;
   24:                 else
   25:                         return false;
   26:         }*/
   27:
   28:         std::shared_ptr<Generator> Generator::Create(std::shared_ptr<EntryPoint> entryPoint)
   29:         {
   30:                 return std::make_shared<Generator>(entryPoint);
   31:         }
   32:
   33:         std::string Generator::GenerateCode(int& errorCode)
   34:         {
   35:                 m_Code = "import libsplw\nscope_vars = libsplw.default_scope.copy()\n\n";
   36:
   37:                 if (m_EntryPoint->require && m_EntryPoint->require->requireType->GetContents() == "trans
piler_py"){
   38:                         m_ScopeImportConfigs.push_back(new ImportConfig(true, true, true, true));
   39:                 }
   40:                 else if (m_EntryPoint->requirePresent == false || m_EntryPoint->require->requireType->Ge
tContents() == "transpiler") {
   41:                         m_ScopeImportConfigs.push_back(new ImportConfig(true, false, false, true));
   42:                 }
   43:                 else if (m_EntryPoint->require) {
   44:                         SPLW_CRITICAL("This generator is not compatible with the specified require state
ment, exiting.");
   45:                         errorCode = 1;
   46:                         return m_Code;
   47:                 }
   48:
   49:                 GenerateStatements(m_EntryPoint->statements);
   50:
   51:                 if (m_AbortPrint) {
   52:                         errorCode = 1;
   53:                 }
   54:
   55:                 return m_Code;
   56:         }
   57:
   58:         void Generator::GenerateStatements(std::shared_ptr<Statements> statements)
   59:         {
   60:                 for (std::shared_ptr<Statement> s : statements->statements)
   61:                 {
   62:                         ImportConfig *importConfig = getCurrentImportConfig();
   63:                         //SPLW_INFO("{0}, {1}, {2}, {3}", importConfig->allowImport, importConfig->allow
PyImport, importConfig->allowPyImport, importConfig->allowBare);
   64:                         switch (s->statementType)
   65:                         {
   66:                         case 0:  GenerateIf(s->ifNode); break; //Line numbers done
   67:                         case 1:  GenerateSet(s->setNode); m_Code += "  # Source line " + std::to_string(
s->lineNumber+1) + "\n"; break;
   68:                         case 2:  GenerateInput(s->inputNode); m_Code += "  # Source line " + std::to_str
ing(s->lineNumber+1) + "\n"; break;
   69:                         case 3:  GenerateOutput(s->outputNode); m_Code += "  # Source line " + std::to_s
tring(s->lineNumber+1) + "\n"; break;
   70:                         case 4:  GenerateInc(s->incNode); m_Code += "  # Source line " + std::to_string(
s->lineNumber+1) + "\n"; break;
   71:                         case 5:  GenerateDec(s->decNode); m_Code += "  # Source line " + std::to_string(
s->lineNumber+1) + "\n"; break;
   72:                         case 6:  GenerateFor(s->forNode); break; // Line numbers done
   73:                         case 7:  GenerateWhile(s->whileNode); break; // Line numbers done
   74:                         case 8:  GenerateQuit(s->quitNode); m_Code += "  # Source line " + std::to_strin
g(s->lineNumber+1) + "\n"; break;
   75:                         case 9:  GenerateCall(s->callNode, true); m_Code += "  # Source line " + std::to
_string(s->lineNumber+1) + "\n"; break;
```

```
   76:                         case 10: GenerateFunc(s->funcNode); break; // Line numbers done
   77:                         case 11: GenerateProc(s->procNode); break; // Line numbers done
   78:                         case 12: GenerateStruct(s->structNode); m_Code += "  # Source line " + std::to_s
tring(s->lineNumber+1) + "\n"; break;
   79:                         case 13: GenerateReturn(s->returnNode); m_Code += "  # Source line " + std::to_s
tring(s->lineNumber+1) + "\n"; break;
   80:                         case 14: GenerateImport(s->importNode); m_Code += "  # Source line " + std::to_s
tring(s->lineNumber+1) + "\n"; break;
   81:                         case 15: importConfig->allowImport = false; importConfig->allowPyImport = false;
 break;
   82:                         case 16: importConfig->allowInstall = false; break;
   83:                         case 17: importConfig->allowBare = false; break;
   84:                         }
   85:                         m_Code += "\n";
   86:                         m_InterpreterCall = false;
   87:                 }
   88:         }
   89:
   90:         void Generator::GenerateIf(std::shared_ptr<IfNode> node)
   91:         {
   92:                 for (uint32_t i = 0; i < node->conditions.size(); i++)
   93:                 {
   94:                         if (i == 0)
   95:                         {
   96:                                 m_Code += m_Tabs + "if ";
   97:                         }
   98:                         else
   99:                         {
  100:                                 m_Code += m_Tabs + "elif ";
  101:                         }
  102:
  103:                         GenerateList(node->conditions.at(i));
  104:                         m_Code += ":   # Source line " + std::to_string(node->lineNumbers[i] + 1) + "\n"
;
  105:
  106:                         m_Tabs += "    ";
  107:
  108:                         GenerateStatements(node->bodies.at(i));
  109:
  110:                         m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
  111:                 }
  112:
  113:                 if (node->elsePresent)
  114:                 {
  115:                         m_Code += m_Tabs + "else: # Source line " + std::to_string(node->lineNumbers[nod
e->lineNumbers.size() - 1] + 1) + "\n";
  116:
  117:                         m_Tabs += "    ";
  118:
  119:                         GenerateStatements(node->bodies.back());
  120:
  121:                         m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
  122:                 }
  123:         }
  124:
  125:         void Generator::GenerateSet(std::shared_ptr<SetNode> node)
  126:         {
  127:                 if (node->id->accessPresent) {
  128:                         bool interpreter_var = false;
  129:                         std::string getattrTree = node->id->GenerateGetattrTree(getCurrentImportConfig()
, interpreter_var, true);
  130:                         if (!interpreter_var) {
  131:                                 m_Code += m_Tabs + "setattr(" + getattrTree + ", \"" + node->id->GetFina
lId() + "\", "; GenerateList(node->list); m_Code += ")";
  132:                         }
  133:                         else {
  134:                                 m_Code += m_Tabs + getattrTree + " = "; GenerateList(node->list);
  135:                         }
  136:                 }
  137:                 else {
  138:                         m_Code += m_Tabs + "scope_vars[\"" + node->id->GetContents() + "\"] = ";  Genera
teList(node->list);
  139:                 }
  140:         }
  141:
  142:         void Generator::GenerateInput(std::shared_ptr<InputNode> node)
  143:         {
  144:                 if (node->id->accessPresent) {
  145:                         bool interpreter_var = false;
  146:                         std::string getattrTree = node->id->GenerateGetattrTree(getCurrentImportConfig()
, interpreter_var, true);
  147:                         if (!interpreter_var) {
  148:                                 m_Code += m_Tabs + "setattr(" + getattrTree + ", \"" + node->id->GetFina
lId() + "\", libsplw.handle_input('";
  149:                         }
  150:                         else {
```

```
  151:                                    m_Code += m_Tabs + getattrTree + " = libsplw.handle_input('";
  152:                            }
  153:                            if (node->signSpec)
  154:                                    m_Code += node->signSpec->GetContents() + " ";
  155:                            GenerateType(node->type);
  156:                            m_Code += "')";
  157:                    } else {
  158:                            m_Code += m_Tabs + "scope_vars[\"" + node->id->GetContents() + "\"] = libsplw.ha
ndle_input('";
  159:                            if (node->signSpec)
  160:                                    m_Code += node->signSpec->GetContents() + " ";
  161:                            GenerateType(node->type);
  162:                            m_Code += "')";
  163:                    }
  164:            }
  165:
  166:            void Generator::GenerateOutput(std::shared_ptr<OutputNode> node)
  167:            {
  168:                    m_Code += m_Tabs + "print(" + ParseRaw(node->raw) + ")";
  169:            }
  170:
  171:            void Generator::GenerateInc(std::shared_ptr<IncNode> node)
  172:            {
  173:                    if (node->id->accessPresent) {
  174:                            bool interpreter_var = false;
  175:                            std::string getAttrTree = node->id->GenerateGetattrTree(getCurrentImportConfig()
, interpreter_var, true);
  176:                            if (interpreter_var) {
  177:                                    m_Code += m_Tabs + getAttrTree + " += 1";
  178:                            }
  179:                            else {
  180:                                    m_Code += m_Tabs + "setattr(" + getAttrTree + ", \"" + node->id->GetFina
lId() + "\", " + node->id->GenerateGetattrTree(getCurrentImportConfig()) + " + 1)";
  181:                            }
  182:                    } else {
  183:                            m_Code += m_Tabs + "scope_vars[\"" + node->id->GetContents() + "\"] += 1";
  184:                    }
  185:            }
  186:
  187:            void Generator::GenerateDec(std::shared_ptr<DecNode> node)
  188:            {
  189:                    if (node->id->accessPresent) {
  190:                            bool interpreter_var = false;
  191:                            std::string getAttrTree = node->id->GenerateGetattrTree(getCurrentImportConfig()
, interpreter_var, true);
  192:                            if (interpreter_var) {
  193:                                    m_Code += m_Tabs + getAttrTree + " -= 1";
  194:                            } else {
  195:                                    m_Code += m_Tabs + "setattr(" + getAttrTree + ", \"" + node->id->GetFina
lId() + "\", " + node->id->GenerateGetattrTree(getCurrentImportConfig()) + " - 1)";
  196:                            }
  197:                    } else {
  198:                            m_Code += m_Tabs + "scope_vars[\"" + node->id->GetContents() + "\"] -= 1";
  199:                    }
  200:            }
  201:
  202:            void Generator::GenerateFor(std::shared_ptr<ForNode> node)
  203:            {
  204:                    if (node->id->accessPresent) {
  205:                            bool interpreter_var = false;
  206:                            std::string getAttrTree = node->id->GenerateGetattrTree(getCurrentImportConfig()
, interpreter_var, true);
  207:                            if (interpreter_var) {
  208:                                    m_Code += m_Tabs + "for " + getAttrTree + " in "; GenerateList(node->ite
rableExpr); m_Code += ": # Source line " + std::to_string(node->lineNumber + 1) + "\n";
  209:                            } else {
  210:                                    std::string for_var = "__for_var_line_" + std::to_string(node->id->GetLi
neNumber()) + "_char_" + std::to_string(node->id->GetColumnNumber());
  211:                                    m_Code += m_Tabs + "for " + for_var + " in "; GenerateList(node->iterabl
eExpr); m_Code += ": # Source line " + std::to_string(node->lineNumber + 1) + "\n";
  212:                                    m_Code += m_Tabs + "    setattr(" + getAttrTree + ", " + node->id->GetFi
nalId() + ", " + for_var + ")";
  213:                            }
  214:                    }
  215:                    else {
  216:                            std::string for_var = "__for_var_line_" + std::to_string(node->id->GetLineNumber
()) + "_char_" + std::to_string(node->id->GetColumnNumber());
  217:                            m_Code += m_Tabs + "for " + for_var + " in "; GenerateList(node->iterableExpr);
m_Code += ": # Source line " + std::to_string(node->lineNumber + 1) + "\n";
  218:                            m_Code += m_Tabs + "    scope_vars[\"" + node->id->GetContents() + "\"] = " + fo
r_var + "\n";
  219:                    }
  220:                    m_Tabs += "    ";
  221:
  222:                    GenerateStatements(node->body);
  223:
```

```
224:                    m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
225:            }
226:
227:            void Generator::GenerateWhile(std::shared_ptr<WhileNode> node)
228:            {
229:                    m_Code += m_Tabs + "while "; GenerateBinOp(node->condition); m_Code += ": # Source line
" + std::to_string(node->lineNumber + 1) + "\n";
230:                    m_Tabs += "    ";
231:
232:                    GenerateStatements(node->body);
233:
234:                    m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
235:            }
236:
237:            void Generator::GenerateQuit(std::shared_ptr<QuitNode> node)
238:            {
239:                    m_Code += m_Tabs + "exit(";
240:                    if (node->returnVal)
241:                            GenerateAtom(node->returnVal);
242:                    m_Code += ")";
243:            }
244:
245:            void Generator::GenerateCall(std::shared_ptr<CallNode> node, bool statement)
246:            {
247:                    if (statement)
248:                            m_Code += m_Tabs;
249:                    GenerateExpr(node->function);
250:                    if (!m_InterpreterCall) {
251:                            m_Code += "(scope_vars";
252:
253:                            if (node->args.size() != 0) {
254:                                    for (uint32_t i = 0; i < node->args.size(); i++) {
255:                                            m_Code += ", "; GenerateExpr(node->args.at(i));
256:                                    }
257:                            }
258:                    }
259:                    else if (node->args.size() != 0){
260:                            m_Code += "("; GenerateExpr(node->args.at(0));
261:
262:                            for (uint32_t i = 1; i < node->args.size(); i++) {
263:                                    m_Code += ", "; GenerateExpr(node->args.at(i));
264:                            }
265:
266:                    }
267:                    else {
268:                            m_Code += "(";
269:                    }
270:                    m_Code += ")";
271:            }
272:
273:            void Generator::GenerateFunc(std::shared_ptr<FuncNode> node)
274:            {
275:                    std::string func_name = "__func_name_line_" + std::to_string(node->id->GetLineNumber())
+ "_char_" + std::to_string(node->id->GetColumnNumber());
276:                    m_Code += m_Tabs + "@libsplw.type_check()\ndef " + func_name + "(prev_scope_vars: dict";
277:
278:                    assert(node->argNames.size() == node->argTypes.size());
279:
280:                    for (uint32_t i = 0; i < node->argNames.size(); i++)
281:                    {
282:                            m_Code += ", arg" + std::to_string(i) + ": ";
283:                            GenerateType(node->argTypes.at(i));
284:                            //node->argNames.at(i)->GetContents()
285:                    }
286:
287:                    m_Code += ") -> ";
288:                    GenerateType(node->returnType);
289:                    m_Code += ": # Source line " + std::to_string(node->lineNumber + 1) + "\n";
290:
291:                    m_Tabs += "    ";
292:                    ImportConfig *oldConfig = getCurrentImportConfig();
293:                    ImportConfig *newConfig = new ImportConfig(oldConfig->allowImport, oldConfig->allowPyImp
ort, oldConfig->allowInstall, oldConfig->allowBare);
294:                    m_ScopeImportConfigs.push_back(newConfig);
295:                    //SPLW_INFO("Current allow bare state: {0}", getCurrentImportConfig()->allowBare);
296:
297:                    m_Code += m_Tabs + "scope_vars = prev_scope_vars.copy()\n";
298:                    for (uint32_t i = 0; i < node->argNames.size(); i++) {
299:                            if (node->argNames[i]->accessPresent) {
300:                                    SPLW_ERROR("Attempting to pass an argument name with an access present.
This is not allowed. Line: {0}, arg: {1}", node->argNames[i]->GetLineNumber(), node->argNames[i]->GetContents()
);
301:                                    m_AbortPrint = true;
302:                            }
303:                            else {
304:                                    m_Code += m_Tabs + "scope_vars['" + node->argNames[i]->GetContents() + "
```

```
’] = arg" + std::to_string(i) + "\n";
  305:                          }
  306:                  }
  307:
  308:                  GenerateStatements(node->body);
  309:                  m_Code += m_Tabs + "raise libsplw.FunctionEndError\n";
  310:                  m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
  311:                  if (node->id->accessPresent) {
  312:                          SPLW_ERROR("Attempting to define a function with an access in the function name.
 This is not allowed. Line: {0}, Name: {1}", node->id->GetLineNumber(), node->id->GetContents());
  313:                          m_AbortPrint = true;
  314:                  }
  315:                  else {
  316:                          m_Code += m_Tabs + "scope_vars[’" + node->id->GetContents() + "’] = " + func_nam
e + "\n";
  317:                  }
  318:                  m_ScopeImportConfigs.pop_back();
  319:          }
  320:
  321:          void Generator::GenerateProc(std::shared_ptr<ProcNode> node)
  322:          {
  323:                  std::string func_name = "__func_name_line_" + std::to_string(node->id->GetLineNumber())
+ "_char_" + std::to_string(node->id->GetColumnNumber());
  324:                  m_Code += m_Tabs + "@libsplw.type_check()\ndef " + func_name + "(prev_scope_vars: dict";
  325:
  326:                  assert(node->argNames.size() == node->argTypes.size());
  327:
  328:                  for (uint32_t i = 0; i < node->argNames.size(); i++)
  329:                  {
  330:                          m_Code += ", arg" + std::to_string(i) + ": ";
  331:                          GenerateType(node->argTypes.at(i));
  332:                          //node->argNames.at(i)->GetContents()
  333:                  }
  334:
  335:                  m_Code += "): # Source line " + std::to_string(node->lineNumber + 1) + "\n";
  336:
  337:                  m_Tabs += "    ";
  338:                  ImportConfig *oldConfig = getCurrentImportConfig();
  339:                  ImportConfig *newConfig = new ImportConfig(oldConfig->allowImport, oldConfig->allowPyImp
ort, oldConfig->allowInstall, oldConfig->allowBare);            m_ScopeImportConfigs.push_back(newConfig);
  340:
  341:                  m_Code += m_Tabs + "scope_vars = prev_scope_vars.copy()\n";
  342:                  for (uint32_t i = 0; i < node->argNames.size(); i++) {
  343:                          if (node->argNames[i]->accessPresent) {
  344:                                  SPLW_ERROR("Attempting to pass an argument name with an access present.
This is not allowed. Line: {0}, arg: {1}", node->argNames[i]->GetLineNumber(), node->argNames[i]->GetContents()
);
  345:                                  m_AbortPrint = true;
  346:                          }
  347:                          else {
  348:                                  m_Code += m_Tabs + "scope_vars[’" + node->argNames[i]->GetContents() + "
’] = arg" + std::to_string(i) + "\n";
  349:                          }
  350:                  }
  351:
  352:                  GenerateStatements(node->body);
  353:                  m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
  354:                  if (node->id->accessPresent) {
  355:                          SPLW_ERROR("Attempting to define a procedure with an access in the procedure nam
e. This is not allowed. Line: {0}, Name: {1}", node->id->GetLineNumber(), node->id->GetContents());
  356:                          m_AbortPrint = true;
  357:                  }
  358:                  else {
  359:                          m_Code += m_Tabs + "scope_vars[’" + node->id->GetContents() + "’] = " + func_nam
e + "\n";
  360:                  }
  361:                  m_ScopeImportConfigs.pop_back();
  362:          }
  363:
  364:          void Generator::GenerateStruct(std::shared_ptr<StructNode> node)
  365:          {
  366:                  //scope_vars[’A’] = libsplw.make_struct_class((’x’, ’y’, ’z’), {’x’:int, ’y’:int, ’z’:in
t}, ’A’)
  367:                  if (node->id->accessPresent) {
  368:                          SPLW_ERROR("Attempting to define a struct with an access in the struct name. Thi
s is not allowed. Line: {0}, Name: {1}", node->id->GetLineNumber(), node->id->GetContents());
  369:                          m_AbortPrint = true;
  370:                  }
  371:                  if (node->names.size() != node->types.size()) {
  372:                          SPLW_CRITICAL("Mismatched types and names in struct declaration. This should not
 be possible. Struct beginning line: {0}", node->id->GetLineNumber());
  373:                          m_AbortPrint = true;
  374:                  }
  375:                  m_Code += m_Tabs + "scope_vars[’" + node->id->GetContents() + "’] = libsplw.make_struct_
class((";
  376:
```

```
377:                if (node->names.size() != 0) {
378:                        m_Code += "'" + node->names[0]->GetContents() + "'";
379:                }
380:                else {
381:                        SPLW_WARN("Structure has no members. This should not be necessary. Structure beg
inning Line: {0}, Name: {1}", node->id->GetLineNumber(), node->id->GetContents());
382:                }
383:                for (uint32_t i = 1; i < node->names.size(); i++) {
384:                        m_Code += ", '" + node->names[i]->GetContents() + "'";
385:                }
386:
387:                m_Code += "), {";
388:
389:                if (node->names.size() != 0) {
390:                        m_Code += "'" + node->names.at(0)->GetContents() + "': ";
391:                        GenerateType(node->types.at(0));
392:                }
393:                for (uint32_t i = 1; i < node->names.size(); i++)
394:                {
395:                        m_Code += ", '" + node->names.at(i)->GetContents() + "': ";
396:                        GenerateType(node->types.at(i));
397:                }
398:                m_Code += "}, '" + node->id->GetContents() + "')";
399:        }
400:
401:        void Generator::GenerateReturn(std::shared_ptr<ReturnNode> node)
402:        {
403:                m_Code += m_Tabs + "return "; GenerateList(node->list);
404:        }
405:
406:        void Generator::GenerateImport(std::shared_ptr<ImportNode> node)
407:        {
408:                m_Code += m_Tabs + "import " + node->id->GetContents();
409:        }
410:
411:        void Generator::GenerateList(std::shared_ptr<ListNode> node, bool fromAtom)
412:        {
413:                if (node->Items.at(0)->hasRight)
414:                {
415:                        m_Code += "{";
416:                }
417:                else if (node->Items.size() > 1)
418:                {
419:                        m_Code += "[";
420:                }
421:                else
422:                {
423:                        if (fromAtom) {
424:                                m_Code += "(";
425:                        }
426:                        GenerateDictEntry(node->Items.at(0));
427:                        if (fromAtom){
428:                                m_Code += ")";
429:                        }
430:                        return;
431:                }
432:
433:                GenerateDictEntry(node->Items.at(0));
434:                for (uint32_t i = 1; i < node->Items.size(); i++)
435:                {
436:                        m_Code += ", ";
437:                        GenerateDictEntry(node->Items.at(i));
438:                }
439:
440:                if (node->Items.at(0)->hasRight)
441:                {
442:                        m_Code += "}";
443:                }
444:                else if (node->Items.size() > 1)
445:                {
446:                        m_Code += "]";
447:                }
448:        }
449:
450:        void Generator::GenerateDictEntry(std::shared_ptr<DictEntryNode> node)
451:        {
452:                GenerateExpr(node->left);
453:                if (node->hasRight)
454:                {
455:                        m_Code.append(": ");
456:                        GenerateExpr(node->right);
457:                }
458:        }
459:
460:        void Generator::GenerateExpr(std::shared_ptr<Expr> node)
461:        {
```

```
462:                switch (node->exprType)
463:                {
464:                case 1: GenerateBinOp(node->binOpNode); break;
465:                case 2: GenerateCreate(node->createNode); break;
466:                case 3: GenerateCast(node->castNode); break;
467:                case 4: GenerateCall(node->callNode, false); break;
468:                case 5: GenerateAnonf(node->anonfNode); break;
469:                case 6: GenerateAnonp(node->anonpNode); break;
470:                }
471:        }
472:
473:        void Generator::GenerateBinOp(std::shared_ptr<BinOpNode> node)
474:        {
475:                GenerateFactor(node->left);
476:                if (node->opToken)
477:                {
478:                        std::string opTokenStr = "";
479:                        //std::transform(opTokenStr.begin(), opTokenStr.end(), opTokenStr.begin(),
480:                        //      [](unsigned char c) { return std::tolower(c); });
481:                        switch (node->opToken->GetType()) {
482:                        case TokenType::Is:
483:                                opTokenStr = "is"; break;
484:                        case TokenType::Not:
485:                                opTokenStr = "not"; break;
486:                        case TokenType::And:
487:                                opTokenStr = "and"; break;
488:                        case TokenType::Or:
489:                                opTokenStr = "or"; break;
490:                        case TokenType::Equal:
491:                                opTokenStr = "=="; break;
492:                        case TokenType::NotEqual:
493:                                opTokenStr = "!="; break;
494:                        case TokenType::LessThan:
495:                                opTokenStr = "<"; break;
496:                        case TokenType::GreaterThan:
497:                                opTokenStr = ">"; break;
498:                        case TokenType::LessThanEqual:
499:                                opTokenStr = "<="; break;
500:                        case TokenType::GreaterThanEqual:
501:                                opTokenStr = ">="; break;
502:                        case TokenType::Multiply:
503:                                opTokenStr = "*"; break;
504:                        case TokenType::Divide:
505:                                opTokenStr = "/"; break;
506:                        case TokenType::Intdiv:
507:                                opTokenStr = "//"; break;
508:                        case TokenType::Plus:
509:                                opTokenStr = "+"; break;
510:                        case TokenType::Minus:
511:                                opTokenStr = "-"; break;
512:                        case TokenType::Modulo:
513:                                opTokenStr = "%"; break;
514:                        case TokenType::Xor:
515:                                opTokenStr = "^"; break;
516:                        case TokenType::BitwiseAnd:
517:                                opTokenStr = "&"; break;
518:                        case TokenType::BitwiseOr:
519:                                opTokenStr = "|"; break;
520:                        case TokenType::ShiftRight:
521:                                opTokenStr = ">>"; break;
522:                        case TokenType::ShiftLeft:
523:                                opTokenStr = "<<"; break;
524:                        default:
525:                                SPLW_CRITICAL("Bug: Operator {0} not handled", node->opToken->GetContent
s());
526:                        }
527:                        m_Code += " " + opTokenStr + " ";
528:                        GenerateBinOp(node->right);
529:                }
530:        }
531:
532:        void Generator::GenerateFactor(std::shared_ptr<FactorNode> node)
533:        {
534:                if (node->opTokenPresent)
535:                {
536:                        std::string opTokenStr = "";
537:                        //std::transform(opTokenStr.begin(), opTokenStr.end(), opTokenStr.begin(),
538:                        //      [](unsigned char c) { return std::tolower(c); });
539:                        switch (node->opToken->GetType()) {
540:                        case TokenType::Is:
541:                                opTokenStr = "is"; break;
542:                        case TokenType::Not:
543:                                opTokenStr = "not"; break;
544:                        case TokenType::And:
545:                                opTokenStr = "and"; break;
546:                        case TokenType::Or:
```

```
547:                                    opTokenStr = "or"; break;
548:                            case TokenType::Equal:
549:                                    opTokenStr = "=="; break;
550:                            case TokenType::NotEqual:
551:                                    opTokenStr = "!="; break;
552:                            case TokenType::LessThan:
553:                                    opTokenStr = "<"; break;
554:                            case TokenType::GreaterThan:
555:                                    opTokenStr = ">"; break;
556:                            case TokenType::LessThanEqual:
557:                                    opTokenStr = "<="; break;
558:                            case TokenType::GreaterThanEqual:
559:                                    opTokenStr = ">="; break;
560:                            case TokenType::Multiply:
561:                                    opTokenStr = "*"; break;
562:                            case TokenType::Divide:
563:                                    opTokenStr = "/"; break;
564:                            case TokenType::Intdiv:
565:                                    opTokenStr = "//"; break;
566:                            case TokenType::Plus:
567:                                    opTokenStr = "+"; break;
568:                            case TokenType::Minus:
569:                                    opTokenStr = "-"; break;
570:                            case TokenType::Modulo:
571:                                    opTokenStr = "%"; break;
572:                            case TokenType::Xor:
573:                                    opTokenStr = "^"; break;
574:                            case TokenType::BitwiseAnd:
575:                                    opTokenStr = "&"; break;
576:                            case TokenType::BitwiseOr:
577:                                    opTokenStr = "|"; break;
578:                            case TokenType::ShiftRight:
579:                                    opTokenStr = ">>"; break;
580:                            case TokenType::ShiftLeft:
581:                                    opTokenStr = "<<"; break;
582:                            default:
583:                                    SPLW_CRITICAL("Bug: Operator {0} not handled", node->opToken->GetContent
s());
584:                            }
585:                            m_Code += " " + opTokenStr + " ";
586:                    }
587:                    GenerateAtom(node->right);
588:            }
589:
590:            void Generator::GenerateAtom(std::shared_ptr<AtomNode> node)
591:            {
592:                    switch (node->type)
593:                    {
594:                    case 1:
595:                    {
596:                            if (node->token->GetType() == TokenType::Raw)
597:                                    m_Code += ParseRaw(node->token);
598:                            else if (node->token->GetType() == TokenType::String)
599:                                    m_Code += "\"" + node->token->GetContents() + "\"";
600:                            else if (node->token->GetType() == TokenType::Complex)
601:                                    m_Code += ParseComplex(node->token);
602:                            else if (node->token->GetType() == TokenType::True)
603:                                    m_Code += "True";
604:                            else if (node->token->GetType() == TokenType::False)
605:                                    m_Code += "False";
606:                            else
607:                                    m_Code += StripLeadingZeros(node->token->GetContents());
608:                            break;
609:                    }
610:                    case 2: GenerateList(node->list, true); break;
611:                    case 3: m_Code += node->ident->GenerateGetattrTree(getCurrentImportConfig(), m_Interpret
erCall, false); break;
612:                    }
613:
614:                    if (node->listAccessPresent)
615:                    {
616:                            for (std::shared_ptr<ListNode> n : node->listAccess->indices)
617:                            {
618:                                    m_Code += "[";
619:                                    GenerateList(n);
620:                                    m_Code += "]";
621:                            }
622:                    }
623:            }
624:
625:            void Generator::GenerateCreate(std::shared_ptr<CreateNode> node)
626:            {
627:                    GenerateType(node->createType); m_Code += "(";
628:
629:                    if (node->args.size() > 0) {
630:                            GenerateExpr(node->args.at(0));
```

```
631:                        for (uint32_t i = 1; i < node->args.size(); i++) {
632:                                m_Code += ", "; GenerateExpr(node->args.at(i));
633:                        }
634:                }
635:
636:                m_Code += ")";
637:        }
638:
639:        void Generator::GenerateCast(std::shared_ptr<CastNode> node)
640:        {
641:                GenerateType(node->castType); m_Code += "("; GenerateList(node->list); m_Code += ")";
642:        }
643:
644:        void Generator::GenerateAnonf(std::shared_ptr<AnonfNode> node)
645:        {
646:                uint32_t charIndex = m_Code.size() - 1;
647:                std::string code = "";
648:                while (m_Code.at(charIndex) != '\n' && charIndex != -1)
649:                {
650:                        code = m_Code.back() + code;
651:                        m_Code.pop_back();
652:                        charIndex--;
653:                }
654:
655:                std::string anonf_name = "__anonf_line_" + std::to_string(node->argNames.at(0)->GetLineN
umber()) + "_" + std::to_string(node->argNames[0]->GetColumnNumber());
656:                m_Code += m_Tabs + "@libsplw.type_check()\ndef " + anonf_name + "(prev_scope_vars: dict"
;
657:
658:                assert(node->argNames.size() == node->argTypes.size());
659:
660:                for (uint32_t i = 0; i < node->argNames.size(); i++)
661:                {
662:                        m_Code += ", arg" + std::to_string(i) + ": ";
663:                        GenerateType(node->argTypes.at(i));
664:                        //node->argNames.at(i)->GetContents()
665:                }
666:
667:                m_Code += ") -> ";
668:                GenerateType(node->returnType);
669:                m_Code += ":\n";
670:
671:                m_Tabs += "    ";
672:                ImportConfig *oldConfig = getCurrentImportConfig();
673:                ImportConfig *newConfig = new ImportConfig(oldConfig->allowImport, oldConfig->allowPyImp
ort, oldConfig->allowInstall, oldConfig->allowBare);
674:                m_ScopeImportConfigs.push_back(newConfig);
675:
676:                m_Code += m_Tabs + "scope_vars = prev_scope_vars.copy()\n";
677:                for (uint32_t i = 0; i < node->argNames.size(); i++) {
678:                        if (node->argNames[i]->accessPresent) {
679:                                SPLW_ERROR("Attempting to pass an argument name with an access present.
This is not allowed. Line: {0}, arg: {1}", node->argNames[i]->GetLineNumber(), node->argNames[i]->GetContents()
);
680:                                m_AbortPrint = true;
681:                        }
682:                        else {
683:                                m_Code += m_Tabs + "scope_vars['" + node->argNames[i]->GetContents() + "
'] = arg" + std::to_string(i) + "\n";
684:                        }
685:                }
686:
687:                GenerateStatements(node->body);
688:                m_Code += m_Tabs + "raise libsplw.FunctionEndError\n";
689:                m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
690:
691:                m_Code += code + anonf_name;
692:                m_ScopeImportConfigs.pop_back();
693:        }
694:
695:        void Generator::GenerateAnonp(std::shared_ptr<AnonpNode> node)
696:        {
697:                uint32_t charIndex = m_Code.size() - 1;
698:                std::string code = "";
699:                while (m_Code.at(charIndex) != '\n' && charIndex != -1)
700:                {
701:                        code = m_Code.back() + code;
702:                        m_Code.pop_back();
703:                        charIndex--;
704:                }
705:
706:                std::string anonp_name = "__anonp_line_" + std::to_string(node->argNames.at(0)->GetLineN
umber()) + "_" + std::to_string(node->argNames[0]->GetColumnNumber());
707:                m_Code += m_Tabs + "@libsplw.type_check()\ndef " + anonp_name + "(prev_scope_vars: dict"
;
708:
```

```
  709:                 assert(node->argNames.size() == node->argTypes.size());
  710:
  711:                 for (uint32_t i = 0; i < node->argNames.size(); i++)
  712:                 {
  713:                         m_Code += ", arg" + std::to_string(i) + ": ";
  714:                         GenerateType(node->argTypes.at(i));
  715:                         //node->argNames.at(i)->GetContents()
  716:                 }
  717:
  718:                 m_Code += "):\n";
  719:
  720:                 m_Tabs += "    ";
  721:                 ImportConfig *oldConfig = getCurrentImportConfig();
  722:                 ImportConfig *newConfig = new ImportConfig(oldConfig->allowImport, oldConfig->allowPyImp
ort, oldConfig->allowInstall, oldConfig->allowBare);
  723:                 m_ScopeImportConfigs.push_back(newConfig);
  724:
  725:                 m_Code += m_Tabs + "scope_vars = prev_scope_vars.copy()\n";
  726:                 for (uint32_t i = 0; i < node->argNames.size(); i++) {
  727:                         if (node->argNames[i]->accessPresent) {
  728:                                 SPLW_ERROR("Attempting to pass an argument name with an access present.
This is not allowed. Line: {0}, arg: {1}", node->argNames[i]->GetLineNumber(), node->argNames[i]->GetContents()
);
  729:                                 m_AbortPrint = true;
  730:                         }
  731:                         else {
  732:                                 m_Code += m_Tabs + "scope_vars['" + node->argNames[i]->GetContents() + "
'] = arg" + std::to_string(i) + "\n";
  733:                         }
  734:                 }
  735:
  736:                 GenerateStatements(node->body);
  737:                 m_Tabs = m_Tabs.substr(0, m_Tabs.size() - 4);
  738:
  739:                 m_Code += code + anonp_name;
  740:                 m_ScopeImportConfigs.pop_back();
  741:         }
  742:
  743:         void Generator::GenerateType(std::shared_ptr<TypeNode> node)
  744:         {
  745:                 if (node->type == 1) {
  746:                         m_Code += node->ident->GenerateGetattrTree(getCurrentImportConfig());
  747:                 }
  748:                 else {
  749:                         //m_Code += "__builtins__.";
  750:                         std::string typeTokenStr = node->typeToken->GetContents();
  751:                         std::transform(typeTokenStr.begin(), typeTokenStr.end(), typeTokenStr.begin(),
  752:                                 [](unsigned char c) { return std::tolower(c); });
  753:                         if (typeTokenStr == "string")
  754:                                 typeTokenStr = "str";
  755:                         else if (typeTokenStr == "real")
  756:                                 typeTokenStr = "float";
  757:                         else if (typeTokenStr == "number")
  758:                                 typeTokenStr = "float";
  759:                         else if (typeTokenStr == "integer")
  760:                                 typeTokenStr = "int";
  761:                         else if (typeTokenStr == "dictionary")
  762:                                 typeTokenStr = "dict";
  763:                         else if (typeTokenStr == "mapping")
  764:                                 typeTokenStr = "dict";
  765:                         else if (typeTokenStr == "map")
  766:                                 typeTokenStr = "dict";
  767:
  768:                         m_Code += typeTokenStr;
  769:                 }
  770:         }
  771:
  772:         /*void Generator::GenerateIdent(std::shared_ptr<IdentNode> node)
  773:         {
  774:                 m_Code += node->GetContents();
  775:         }*/
  776:
  777:         bool validIdentifier(std::string id) {
  778:                 std::smatch m;
  779:                 if (std::regex_search(id, m, std::regex("(\\d|_)+(\\.\\d+)?i")) && m[0] == id) // Matche
s complex regex
  780:         {
  781:                         return false;
  782:         }
  783:         else if (std::regex_search(id, m, std::regex("(\\d|_)+\\.\\d+")) && m[0] == id) // Matches float
 regex
  784:         {
  785:             return false;
  786:         }
  787:         else if (std::regex_search(id, m, std::regex("(\\d+_*)+")) && m[0] == id) // Matches int regex
  788:         {
```

```
  789:             return false;
  790:         }
  791:         else
  792:         {
  793:             char invalidChars[] = { '~', '\\', ';', '#', '$', '@', ''', ',', '?', '.', '!', '%', '^', '<
', '|', '\'', '&', ')', '*', '/', '+', '[', ']', '"', '=', '{', '}', ':', '>', '(', '-'};
  794:             bool valid = true;
  795:             int index = 0;
  796:             for (char c : id) {
  797:                 for (char d : invalidChars) {
  798:                     if (c == d) {
  799:                         valid = false;
  800:                         break;
  801:                     }
  802:                 }
  803:                 if (valid == false)
  804:                     break;
  805:                 index++;
  806:             }
  807:             if (valid)
  808:                 return true;
  809:             else
  810:                                 return false;
  811:                 }
  812:         }
  813:
  814:         std::string Generator::ParseRaw(std::shared_ptr<Token> token)
  815:         {
  816:                 std::string code = "fr\"";
  817:                 bool inIdent = false;
  818:                 std::shared_ptr<IdentNode> identNode = std::make_shared<IdentNode>();
  819:                 std::string ident = "";
  820:                 for (char c : token->GetContents())
  821:                 {
  822:                         if (!inIdent && !charInStr("$\"", c))
  823:                                 code += c;
  824:                         else if (inIdent) {
  825:                                 if (c == ' ') {
  826:                                         if (validIdentifier(ident)) {
  827:                                                 identNode->ids.push_back(std::make_shared<Token>(TokenTy
pe::Identifier, ident.c_str(), token->GetLineNumber(), token->GetCharacterNumber()));
  828:                                                 code += identNode->GenerateGetattrTree(getCurrentImportC
onfig());
  829:                                         }
  830:                                         else {
  831:                                                 SPLW_ERROR("Invalid identifier in RAW token, line {0}, c
har {1}", token->GetLineNumber(), token->GetCharacterNumber());
  832:                                                 m_AbortPrint = true;
  833:                                         }
  834:                                         code += "} ";
  835:                                         inIdent = false;
  836:                                         ident = "";
  837:                                         identNode = std::make_shared<IdentNode>();
  838:                                 }
  839:                                 else if (c == '.') {
  840:                                         identNode->ids.push_back(std::make_shared<Token>(TokenType::Iden
tifier, ident.c_str(), token->GetLineNumber(), token->GetCharacterNumber()));
  841:                                         identNode->accessPresent = true;
  842:                                         ident = "";
  843:                                 }
  844:                                 else {
  845:                                         ident += c;
  846:                                 }
  847:                         }
  848:                         else if (c == '"')
  849:                         {
  850:                                 code += "\" + \"\\\"\" + fr\"";
  851:                                 //code += "\\" + c;
  852:                         }
  853:                         else
  854:                         {
  855:                                 inIdent = true;
  856:                                 code += "{";
  857:                         }
  858:                 }
  859:                 if (inIdent) {
  860:                         if (validIdentifier(ident)) {
  861:                                 identNode->ids.push_back(std::make_shared<Token>(TokenType::Identifier,
ident.c_str(), token->GetLineNumber(), token->GetCharacterNumber()));
  862:                                 code += identNode->GenerateGetattrTree(getCurrentImportConfig());
  863:                         } else {
  864:                                 SPLW_ERROR("Invalid identifier in RAW token, line {0}, char {1}", token-
>GetLineNumber(), token->GetCharacterNumber());
  865:                                 m_AbortPrint = true;
  866:                         }
  867:                         code += "}";
```

```
868:                 }
869:                 return code + "\"";
870:         }
871:
872:         std::string Generator::ParseComplex(std::shared_ptr<Token> token)
873:         {
874:                 std::string code = "";
875:                 for (char c : token->GetContents())
876:                 {
877:                         if (c == 'i')
878:                                 code += 'j';
879:                         else
880:                                 code += c;
881:                 }
882:                 return StripLeadingZeros(code);
883:         }
884:
885:         std::string Generator::StripLeadingZeros(std::string token) {
886:                 uint64_t EndOfLeadingZeros = 0;
887:                 while (token[EndOfLeadingZeros] == '0' || token[EndOfLeadingZeros] == '_')
888:                         EndOfLeadingZeros++;
889:                 if (token.size() <= EndOfLeadingZeros || token[EndOfLeadingZeros] == '.')
890:                         EndOfLeadingZeros--;
891:                 return token.substr(EndOfLeadingZeros);
892:         }
893: }
```

```
    1: ï»¿#include <cstdint>
    2: #include <string>
    3: #include <iostream>
    4: #include <chrono>
    5:
    6: #ifdef SPLW_WINDOWS
    7: #include <Windows.h>
    8: #endif
    9:
   10: //#include "Instrumentor.h"
   11: #include "Transpiler.h"
   12: #include "Log.h"
   13:
   14: using namespace Spliwaca;
   15:
   16: std::shared_ptr<TranspilerState> state = std::make_shared<TranspilerState>();
   17:
   18: //------------------------------------ UtilFunctions utility function definitions ----------------------
----------
   19: class MissingVariable
   20: {
   21:         uint32_t lineNumber;
   22:         uint32_t columnNumber;
   23:
   24: public:
   25:         MissingVariable(uint32_t lineNumber, uint32_t columnNumber)
   26:                 : lineNumber(lineNumber), columnNumber(columnNumber) {}
   27:
   28:         uint32_t GetLineNumber() const { return lineNumber; }
   29:         uint32_t GetColumnNumber() const { return columnNumber; }
   30:         uint32_t GetColumnSpan() const { return 1; }
   31: };
   32:
   33: int RegisterLexicalError(uint8_t errorCode, uint32_t lineNumber, uint32_t columnNumber, uint16_t columnS
pan)
   34: {
   35:         state->LexerErrors.push_back({errorCode, lineNumber, columnNumber, columnSpan});
   36:         return 1;
   37: }
   38:
   39: int RegisterSyntaxError(SyntaxErrorType type, std::shared_ptr<Token> token)
   40: {
   41:         state->SyntaxErrors.push_back({type, token});
   42:         return 1;
   43: }
   44:
   45: int RegisterSyntaxError(SyntaxErrorType errorCode, uint32_t lineNumber, uint32_t columnNumber, size_t co
lumnSpan, Spliwaca::TokenType type)
   46: {
   47:         state->SyntaxErrors.push_back({errorCode, lineNumber, columnNumber, columnSpan, type});
   48:         return 1;
   49: }
   50:
   51: int RegisterSemanticsError(uint32_t lineNumber, uint32_t columnNumber)
   52: {
   53:         //state->SemanticErrors.push_back(MissingVariable(lineNumber, columnNumber));
   54:         return 1;
   55: }
   56:
   57: std::string mulString(std::string s, int i)
   58: {
   59:         if (i <= 0)
   60:                 return "";
   61:         std::string init = s;
   62:         for (size_t j = 0; j < i; j++)
   63:         {
   64:                 s.append(init);
   65:         }
   66:         return s;
   67: }
   68:
   69: /* Code snippet copied from https://stackoverflow.com/questions/1489830/efficient-way-to-determine-numbe
r-of-digits-in-an-integer
   70:    accepted answer */
   71: int numDigits(int32_t x)
   72: {
   73:         if (x >= 10000)
   74:         {
   75:                 if (x >= 10000000)
   76:                 {
   77:                         if (x >= 100000000)
   78:                         {
   79:                                 if (x >= 1000000000)
   80:                                         return 10;
   81:                                 return 9;
   82:                         }
```

```
   83:                         return 8;
   84:                 }
   85:                 if (x >= 100000)
   86:                 {
   87:                         if (x >= 1000000)
   88:                                 return 7;
   89:                         return 6;
   90:                 }
   91:                 return 5;
   92:         }
   93:         if (x >= 100)
   94:         {
   95:                 if (x >= 1000)
   96:                         return 4;
   97:                 return 3;
   98:         }
   99:         if (x >= 10)
  100:                 return 2;
  101:         return 1;
  102: }
  103:
  104: bool charInStr(const std::string &s, char c)
  105: {
  106:         //PROFILE_FUNC();
  107:         for (char ch : s)
  108:         {
  109:                 if (ch == c)
  110:                 {
  111:                         return true;
  112:                 }
  113:         }
  114:         return false;
  115: }
  116:
  117: template <typename T>
  118: bool itemInVect(const std::vector<T> &v, T t)
  119: {
  120:         for (T e : v)
  121:         {
  122:                 if (e == t)
  123:                 {
  124:                         return true;
  125:                 }
  126:         }
  127:         return false;
  128: }
  129:
  130: class Timer
  131: {
  132: public:
  133:         Timer() : beg_(clock_::now()) {}
  134:         void reset() { beg_ = clock_::now(); }
  135:         double elapsed() const
  136:         {
  137:                 return std::chrono::duration_cast<second_>(clock_::now() - beg_).count();
  138:         }
  139:
  140: private:
  141:         typedef std::chrono::high_resolution_clock clock_;
  142:         typedef std::chrono::duration<double, std::ratio<1>> second_;
  143:         std::chrono::time_point<clock_> beg_;
  144: };
  145:
  146: //----------------------------------- End UtilFunctions utility function definitions -----------------
--------------
  147:
  148: struct transpilerOptions {
  149:         std::string ifile;
  150:         std::string ofile;
  151:         bool recursive_transpile;
  152: };
  153:
  154: transpilerOptions *parseCommandLineArgs(int argc, char **argv) {
  155:         transpilerOptions *options = new transpilerOptions();
  156:         if (argc < 2) {
  157:                 std::cout << "Usage: transpiler FILE [-o OUTFILE]\n";
  158:                 exit(-1);
  159:         }
  160:         else if (argc > 2 && argc != 4) {
  161:                 std::cout << "Usage: transpiler FILE [-o OUTFILE]\n";
  162:                 exit(-1);
  163:         }
  164:         else if (argc == 4 && strcmp(argv[2], "-o")) {
  165:                 std::cout << "Usage: transpiler FILE [-o OUTFILE]\n";
  166:                 exit(-1);
  167:         }
```

```
  168:         else if (argc == 4) {
  169:                 options->ofile = argv[3];
  170:         }
  171:         else {
  172:                 options->ofile = "";
  173:         }
  174:         options->ifile = argv[1];
  175:         return options;
  176: }
  177:
  178: int main(int argc, char** argv)
  179: {
  180:         transpilerOptions *options = parseCommandLineArgs(argc, argv);
  181:         std::string inFile = options->ifile, outFile = options->ofile;
  182:
  183:         Timer totalTimer = Timer();
  184:
  185:         #ifdef SPLW_WINDOWS
  186:         SetConsoleOutputCP(CP_UTF8);
  187:         setvbuf(stdout, nullptr, _IOFBF, 1000);
  188:         #endif
  189:
  190:         LOG_INIT();
  191:         bool printTokenList = false;
  192:
  193:         Transpiler transpiler = Transpiler(inFile, outFile, state, printTokenList);
  194:         std::string output = transpiler.Run();
  195:
  196:         //std::cout << "\nLexer took: " << lexerTime << " seconds\nParser took: " << parseTime << " seco
nds\nGenerator took: " << generateTime << " seconds" << std::endl;
  197:         std::cout << "#Total time taken: " << totalTimer.elapsed() << std::endl;
  198:
  199:         #ifdef SPLW_WINDOWS
  200:         system("PAUSE");
  201:         #else
  202:                 //system("read -n 1 -s -p \"Press any key to continue...\n\"");
  203:         #endif
  204:         return 0;
  205: }
```

```
    1: #include "Transpiler.h"
    2: #include "Log.h"
    3: #include <iostream>
    4: #include <fstream>
    5:
    6: namespace Spliwaca
    7: {
    8:         std::string Transpiler::Run()
    9:         {
   10:                 //Timer lexerTimer = Timer();
   11:
   12:                 std::shared_ptr<Lexer> lexer = Lexer::Create(m_Filename);
   13:                 SPLW_INFO("Created lexer.");
   14:                 std::shared_ptr<std::vector<std::shared_ptr<Token>>> tokens = lexer->MakeTokens();
   15:
   16:                 for (LexicalError l : m_State->LexerErrors)
   17:                 {
   18:                         SPLW_ERROR("Lexical Error code {2} at line {0}, column {1}", l.GetLineNumber(),
l.GetColumnNumber(), l.GetErrorCode());
   19:                         if (l.GetLineNumber() >= lexer->GetSplitFileString().size())
   20:                                 SPLW_WARN("Line {0} out of range!", l.GetLineNumber());
   21:                         else
   22:                                 SPLW_WARN("{0}", lexer->GetSplitFileString().at(l.GetLineNumber()));
   23:                         SPLW_WARN("{0}{1}", mulString(" ", l.GetColumnNumber() - 1), mulString("^", l.Ge
tColumnSpan()));
   24:                         std::cout << "\n";
   25:                 }
   26:                 if (m_State->LexerErrors.size() > 0)
   27:                 {
   28:                         SPLW_ERROR("Lexical errors present: cannot continue to parsing stage.");
   29:                         system("PAUSE");
   30:                         return "";
   31:                 }
   32:                 else
   33:                         SPLW_INFO("Finished constructing tokens.");
   34:
   35:                 //double lexerTime = lexerTimer.elapsed();
   36:
   37:                 if (m_PrintTokenList)
   38:                 {
   39:                         int i = 0;
   40:                         for (std::shared_ptr<Token> t : *tokens)
   41:                         {
   42:                                 if (t->GetContents() == "\n")
   43:                                         SPLW_TRACE("Token {0}: {1},{2} type: {3}, contents: {4}", i, t->
GetLineNumber(), t->GetCharacterNumber(), TokenTypeName(t->GetType()), "\\n");//, mulString(" ", 3 - std::to_st
ring(i).size()), mulString(" ", numDigits(lineCount) - std::to_string(t->GetLineNumber()).size()), mulString("
", 3 - std::to_string(t->GetCharacterNumber()).size()), mulString(" ", 16 - TokenTypeName(t->GetType()).size())
);
   44:                                 else if (t->GetContents() == "\t")
   45:                                         SPLW_TRACE("Token {0}: {1},{2} type: {3}, contents: {4}", i, t->
GetLineNumber(), t->GetCharacterNumber(), TokenTypeName(t->GetType()), "\\t");//, mulString(" ", 3 - std::to_st
ring(i).size()), mulString(" ", numDigits(lineCount) - std::to_string(t->GetLineNumber()).size()), mulString("
", 3 - std::to_string(t->GetCharacterNumber()).size()), mulString(" ", 16 - TokenTypeName(t->GetType()).size())
);
   46:                                 else if (t->GetContents() == "\f")
   47:                                         SPLW_TRACE("Token {0}: {1},{2} type: {3}, contents: {4}", i, t->
GetLineNumber(), t->GetCharacterNumber(), TokenTypeName(t->GetType()), "\\f");//, mulString(" ", 3 - std::to_st
ring(i).size()), mulString(" ", numDigits(lineCount) - std::to_string(t->GetLineNumber()).size()), mulString("
", 3 - std::to_string(t->GetCharacterNumber()).size()), mulString(" ", 16 - TokenTypeName(t->GetType()).size())
);
   48:                                 else
   49:                                         SPLW_TRACE("Token {0}: {1},{2} type: {3}, contents: {4}", i, t->
GetLineNumber(), t->GetCharacterNumber(), TokenTypeName(t->GetType()), t->GetContents());//, mulString(" ", 3 -
 std::to_string(i).size()), mulString(" ", numDigits(lineCount) - std::to_string(t->GetLineNumber()).size()), m
ulString(" ", 3 - std::to_string(t->GetCharacterNumber()).size()), mulString(" ", 16 - TokenTypeName(t->GetType
()).size()));
   50:                                 i++;
   51:                         }
   52:                 }
   53:
   54:                 //Timer parseTimer = Timer();
   55:
   56:                 std::shared_ptr<Parser> parser = Parser::Create(tokens);
   57:                 SPLW_INFO("Created Parser.");
   58:                 std::shared_ptr<Spliwaca::EntryPoint> ast = parser->ConstructAST();
   59:
   60:                 uint32_t prevLineNumber = 0, prevColNumber = 0;
   61:                 for (SyntaxError s : m_State->SyntaxErrors)
   62:                 {
   63:                         if (s.GetLineNumber() == prevLineNumber && s.GetColumnNumber() == prevColNumber)
   64:                                 continue;
   65:                         else {
   66:                                 prevLineNumber = s.GetLineNumber();
   67:                                 prevColNumber = s.GetColumnNumber();
   68:                         }
```

```
   69:                         SPLW_ERROR("Syntax Error code {2} at line {0}, column {1}", s.GetLineNumber(), s
.GetColumnNumber(), s.GetErrorCode());
   70:                         SPLW_ERROR(GetSyntaxErrorMessage(s.GetErrorCode()), TokenTypeName(s.GetTokenType
()));
   71:                         if (s.GetLineNumber() >= lexer->GetSplitFileString().size())
   72:                                 SPLW_WARN("Line {0} out of range!", s.GetLineNumber());
   73:                         else
   74:                                 SPLW_WARN("{0}", lexer->GetSplitFileString().at(s.GetLineNumber()));
   75:                         SPLW_WARN("{0}{1}", mulString(" ", s.GetColumnNumber() - 1), mulString("^", s.Ge
tColumnSpan()));
   76:                         std::cout << "\n";
   77:                 }
   78:                 if (m_State->SyntaxErrors.size() > 0)
   79:                 {
   80:                         SPLW_ERROR("Syntax errors present: cannot continue to next stage.");
   81:                         //if (m_State->MissingVariables.size() == 0)
   82:                         //{
   83:                         //}
   84:                         #ifdef SPLW_WINDOWS
   85:                         system("PAUSE");
   86:                         #else
   87:                         //system("read -n 1 -s -p \"Press any key to continue...\n\"");
   88:                         #endif
   89:                         return "";
   90:                 }
   91:                 else
   92:                         SPLW_INFO("Finished syntax analysis.");
   93:
   94:                 //double parseTime = parseTimer.elapsed();
   95:
   96:                 //Timer generateTimer = Timer();
   97:
   98:                 std::shared_ptr<Generator> codeGenerator = Generator::Create(ast);
   99:                 SPLW_INFO("Created Generator");
  100:
  101:                 int errorCode = 0;
  102:                 std::string finalCode = codeGenerator->GenerateCode(errorCode);
  103:                 if (errorCode) {
  104:                         SPLW_CRITICAL("Errors detected during generation. Aborting.");
  105:                         return "";
  106:                 }
  107:
  108:                 if (m_Output != "") {
  109:                         std::ofstream outputFile;
  110:                         outputFile.open(m_Output, std::ios::trunc);
  111:                         outputFile << finalCode << "\n";
  112:                         outputFile.close();
  113:                 }
  114:                 else {
  115:                         std::cout << finalCode << std::endl;
  116:                 }
  117:
  118:                 //double generateTime = generateTimer.elapsed();
  119:
  120:                 //std::cout << finalCode << std::endl;
  121:
  122:                 SPLW_INFO("Finished code output!");
  123:
  124:                 return finalCode;
  125:         }
  126: }
```