```
    1: #pragma once
    2: #include "Frontend/Lexer/LexicalError.h"
    3: #include "Frontend/Parser/SyntaxError.h"
    4:
    5: int RegisterLexicalError(uint8_t errorCode, uint32_t lineNumber, uint32_t columnNumber, uint16_t columnS
pan = 1);
    6: int RegisterSyntaxError(Spliwaca::SyntaxErrorType type, std::shared_ptr<Spliwaca::Token> token);
    7: int RegisterSyntaxError(Spliwaca::SyntaxErrorType errorCode, uint32_t lineNumber, uint32_t columnNumber,
 size_t columnSpan, Spliwaca::TokenType type);
    8: int RegisterSemanticsError(uint32_t lineNumber, uint32_t columnNumber);
    9: std::string mulString(std::string s, int i);
   10:
   11: int numDigits(int32_t x);
   12:
   13: bool charInStr(const std::string& s, char c);
   14: //template<typename T>
   15: //bool itemInVect(const std::vector<T>& v, T t);
```

```
    1: ï»¿#pragma once
    2: #include "Token.h"
    3: #include <memory>
    4: #include <vector>
    5: #include <map>
    6: #include <sstream>
    7:
    8: namespace Spliwaca
    9: {
   10:         class Lexer
   11:         {
   12:         public:
   13:                 static std::shared_ptr<Lexer> Create(std::string file);
   14:                 ~Lexer() = default;
   15:
   16:                 std::shared_ptr<std::vector<std::shared_ptr<Token>>> MakeTokens();
   17:                 inline const std::string GetFileString() const { return m_FileString; }
   18:                 inline const std::vector<std::string> GetSplitFileString() const { return split(m_FileSt
ring, '\n'); }
   19:                 inline const bool IsStringInKeywords(std::string string) const { return s_KeywordDict.fi
nd(string) != s_KeywordDict.end(); }
   20:
   21:         private:
   22:                 Lexer(std::string file);
   23:
   24:                 void makeToken(std::string tokenContents);
   25:
   26:                 template <typename Out>
   27:                 static void split(const std::string& s, char delim, Out result)
   28:                 {
   29:                         std::istringstream iss(s);
   30:                         std::string item;
   31:                         while (std::getline(iss, item, delim))
   32:                         {
   33:                                 *result++ = item;
   34:                         }
   35:                 }
   36:
   37:                 static std::vector<std::string> split(const std::string& s, char delim)
   38:                 {
   39:                         std::vector<std::string> elems;
   40:                         split(s, delim, std::back_inserter(elems));
   41:                         return elems;
   42:                 }
   43:
   44:         private:
   45:                 std::string m_FileLocation;
   46:                 std::shared_ptr<std::vector<std::shared_ptr<Token>>> m_Tokens;
   47:
   48:                 uint32_t m_LineNumber = 0, m_ColumnNumber = 0;
   49:                 uint32_t m_StoredLineNumber = 0, m_StoredColumnNumber = 0;
   50:
   51:                 //LEXER STATE FLAGS
   52:                 // double_quote single_quote raw block_comment line_comment
   53:                 char flags = 0;
   54:                 std::string persistent_contents;
   55:
   56:                 std::string m_FileString;
   57:
   58:                 const std::string alphabetCharacters = {
   59:                         "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_"
   60:                 };
   61:
   62:                 const std::map<std::string, TokenType> s_KeywordDict = {
   63:                         {"INT",                 TokenType::Type},
   64:                         {"Int",                 TokenType::Type},
   65:                         {"int",                 TokenType::Type},
   66:                         {"INTEGER",             TokenType::Type},
   67:                         {"Integer",             TokenType::Type},
   68:                         {"integer",             TokenType::Type},
   69:                         {"FLOAT",               TokenType::Type},
   70:                         {"Float",               TokenType::Type},
   71:                         {"float",               TokenType::Type},
   72:                         {"REAL",                TokenType::Type},
   73:                         {"Real",                TokenType::Type},
   74:                         {"real",                TokenType::Type},
   75:                         {"NUMBER",              TokenType::Type},
   76:                         {"Number",              TokenType::Type},
   77:                         {"number",              TokenType::Type},
   78:                         {"COMPLEX",             TokenType::Type},
   79:                         {"Complex",             TokenType::Type},
   80:                         {"complex",             TokenType::Type},
   81:                         {"STRING",              TokenType::Type},
   82:                         {"String",              TokenType::Type},
   83:                         {"string",              TokenType::Type},
   84:                         {"BOOL",                TokenType::Type},
```

```
85:                     {"Bool",              TokenType::Type},
86:                     {"bool",              TokenType::Type},
87:                     {"STR",               TokenType::Type},
88:                     {"Str",               TokenType::Type},
89:                     {"str",               TokenType::Type},
90:                     {"DICT",              TokenType::Type},
91:                     {"Dict",              TokenType::Type},
92:                     {"dict",              TokenType::Type},
93:                     {"DICTIONARY",        TokenType::Type},
94:                     {"Dictionary",        TokenType::Type},
95:                     {"dictionary",        TokenType::Type},
96:                     {"MAP",               TokenType::Type},
97:                     {"Map",               TokenType::Type},
98:                     {"map",               TokenType::Type},
99:                     {"MAPPING",           TokenType::Type},
100:                    {"Mapping",           TokenType::Type},
101:                    {"mapping",           TokenType::Type},
102:                    {"LIST",              TokenType::Type},
103:                    {"List",              TokenType::Type},
104:                    {"list",              TokenType::Type},
105:                    {"ARRAY",             TokenType::Type},
106:                    {"Array",             TokenType::Type},
107:                    {"array",             TokenType::Type},
108:                    {"TUPLE",             TokenType::Type},
109:                    {"Tuple",             TokenType::Type},
110:                    {"tuple",             TokenType::Type},
111:                    {"NULL",              TokenType::None},
112:                    {"Null",              TokenType::None},
113:                    {"null",              TokenType::None},
114:                    {"NONE",              TokenType::None},
115:                    {"None",              TokenType::None},
116:                    {"none",              TokenType::None},
117:                    {"EMPTY",             TokenType::None},
118:                    {"Empty",             TokenType::None},
119:                    {"empty",             TokenType::None},
120:                    {"INFINITY",          TokenType::Float},
121:                    {"Infinity",          TokenType::Float},
122:                    {"infinity",          TokenType::Float},
123:                    {"NAN",               TokenType::Float},
124:                    {"NaN",               TokenType::Float},
125:                    {"nan",               TokenType::Float},
126:                    {"IS",                TokenType::Is},
127:                    {"â\211£",                 TokenType::Is},
128:                    {"â\211¡",                 TokenType::Is},
129:                    {"===",               TokenType::Is},
130:                    {"NOT",               TokenType::Not},
131:                    {"!",                 TokenType::Not},
132:                    {"AND",               TokenType::And},
133:                    {"&&",                TokenType::And},
134:                    {"â\210§",                TokenType::And},
135:                    {"OR",                TokenType::Or},
136:                    {"â\210¨",                TokenType::Or},
137:                    {"||",                TokenType::Or},
138:                    {"=",                 TokenType::Equal},
139:                    {"==",                TokenType::Equal},
140:                    {"EQUALS",            TokenType::Equal},
141:                    {"!=",                TokenType::NotEqual},
142:                    {"â\211 ",                 TokenType::NotEqual},
143:                    {"=/=",               TokenType::NotEqual},
144:                    {"<",                 TokenType::LessThan},
145:                    {"â\211¨",                 TokenType::LessThan},
146:                    {"â\211±",                 TokenType::LessThan},
147:                    {">",                 TokenType::GreaterThan},
148:                    {"â\211©",                 TokenType::GreaterThan},
149:                    {"â\211ª",                 TokenType::GreaterThan},
150:                    {"<=",                TokenType::LessThanEqual},
151:                    {"â\211¤",                 TokenType::LessThanEqual},
152:                    {"â©½",                  TokenType::LessThanEqual},
153:                    {"â\211¦",                  TokenType::LessThanEqual},
154:                    {"â\211¯",                 TokenType::LessThanEqual},
155:                    {"â©¾",                  TokenType::GreaterThanEqual},
156:                    {">=",                TokenType::GreaterThanEqual},
157:                    {"â\211¥",                 TokenType::GreaterThanEqual},
158:                    {"â\211§",                TokenType::GreaterThanEqual},
159:                    {"â\211®",                 TokenType::GreaterThanEqual},
160:                    {"*",                 TokenType::Multiply},
161:                    {"**",                TokenType::Power},
162:                    {"/",                 TokenType::Divide},
163:                    {"DIVI",              TokenType::Intdiv},
164:                    {"+",                 TokenType::Plus},
165:                    {"-",                 TokenType::Minus},
166:                    {"%",                 TokenType::Modulo},
167:                    {"^",                 TokenType::Xor},
168:                    {"&",                 TokenType::BitwiseAnd},
169:                    {"|",                 TokenType::BitwiseOr},
170:                    {">>",                TokenType::ShiftRight},
```

```
171:                        {"<<",                  TokenType::ShiftLeft},
172:                        {"(",                   TokenType::LParen},
173:                        {")",                   TokenType::RParen},
174:                        {"RAW",                 TokenType::Raw},
175:                        {"PLAINTEXT",           TokenType::Raw},
176:                        {"INPUT",               TokenType::Input},
177:                        {"POS",                 TokenType::PositiveTypeMod},
178:                        {"POSITIVE",            TokenType::PositiveTypeMod},
179:                        {"NEG",                 TokenType::NegativeTypeMod},
180:                        {"NEGATIVE",            TokenType::NegativeTypeMod},
181:                        {"NONZERO",             TokenType::NonZeroTypeMod},
182:                        {"NONPOS",              TokenType::NegativeTypeMod},
183:                        {"NONPOSITIVE",         TokenType::NegativeTypeMod},
184:                        {"NONNEG",              TokenType::PositiveTypeMod},
185:                        {"NONNEGATIVE",         TokenType::PositiveTypeMod},
186:                        {"OUTPUT",              TokenType::Output},
187:                        {"PRINT",               TokenType::Output},
188:                        {"CREATE",              TokenType::Create},
189:                        {"CAST",                TokenType::Cast},
190:                        {"CALL",                TokenType::Call},
191:                        {"WITH",                TokenType::With},
192:                        {"QUIT",                TokenType::Quit},
193:                        {"EXIT",                TokenType::Quit},
194:                        {"STOP",                TokenType::Quit},
195:                        {"REQUIRE",             TokenType::Require},
196:                        {"INCREMENT",           TokenType::Increment},
197:                        {"INC",                 TokenType::Increment},
198:                        {"DECREMENT",           TokenType::Decrement},
199:                        {"DEC",                 TokenType::Decrement},
200:                        {"IF",                  TokenType::If},
201:                        {"DO",                  TokenType::Do},
202:                        {"ELSE",                TokenType::Else},
203:                        {"FUNCTION",            TokenType::Function},
204:                        {"FUNC",                TokenType::Function},
205:                        {"TAKES",               TokenType::Takes},
206:                        {"<-",                  TokenType::Takes},
207:                        {"DEFAULT",             TokenType::Default},
208:                        {"RETURNS",             TokenType::ReturnType},
209:                        {"->",                  TokenType::ReturnType},
210:                        {"AS",                  TokenType::As},
211:                        {"RETURN",              TokenType::Return},
212:                        {"PROCEDURE",           TokenType::Procedure},
213:                        {"END",                 TokenType::End},
214:                        {"ANONF",               TokenType::AnonFunc},
215:                        {"ANONFUNC",            TokenType::AnonFunc},
216:                        {"ANONFUNCTION",        TokenType::AnonFunc},
217:                        {"ANONYMOUSF",          TokenType::AnonFunc},
218:                        {"ANONYMOUSFUNC",       TokenType::AnonFunc},
219:                        {"ANONYMOUSFUNCTION",   TokenType::AnonFunc},
220:                        {"ANONP",               TokenType::AnonProc},
221:                        {"ANONPROC",            TokenType::AnonProc},
222:                        {"ANONPROCEDURE",       TokenType::AnonProc},
223:                        {"ANONYMOUSP",          TokenType::AnonProc},
224:                        {"ANONYMOUSPROC",       TokenType::AnonProc},
225:                        {"ANONYMOUSPROCEDURE",  TokenType::AnonProc},
226:                        {"SET",                 TokenType::Set},
227:                        {"TO",                  TokenType::To},
228:                        {"FOR",                 TokenType::For},
229:                        {"OF",                  TokenType::Of},
230:                        {"WHILE",               TokenType::While},
231:                        {"STRUCTURE",           TokenType::Struct},
232:                        {"STRUCT",              TokenType::Struct},
233:                        {"BREAK",               TokenType::Break},
234:                        {"IMPORT",              TokenType::Import},
235:                        {"//",                  TokenType::SingleLineComment},
236:                        {"/*",                  TokenType::StartMultiLineComment},
237:                        {"*/",                  TokenType::EndMultiLineComment},
238:                        {":",                   TokenType::DictEquator},
239:                        {"\n",                  TokenType::Newline},
240:                        {",",                   TokenType::Comma},
241:                        {".",                   TokenType::VarAccessOp},
242:                        {"NOIMPORT",            TokenType::NoImport},
243:                        {"NOINSTALL",           TokenType::NoInstall},
244:                        {"NOBARE",              TokenType::NoBare}
245:                        /*{"\f",                  TokenType::Whitespace},
246:                        //{"\u200b",              TokenType::Whitespace},
247:                        {"\t",                  TokenType::Whitespace},
248:                        {" ",                   TokenType::Whitespace}*/
249:                };
250:        };
251: }
```

```
#pragma once
#include <string>
#include <ostream>

namespace Spliwaca
{
	enum class TokenType
	{
		None = 0,
		Int, //
		Float, //
		Complex, //
		PositiveTypeMod,
		NegativeTypeMod,
		NonZeroTypeMod,
		Increment,
		Decrement,
		String, //
		//Bool,
		True, //
		False,
		Plus,
		Minus,
		Multiply,
		Divide,
		Intdiv,
		Modulo,
		Power,
		Is,
		And,
		Or,
		Not,
		Equal,
		NotEqual,
		LessThan,
		GreaterThan,
		LessThanEqual,
		GreaterThanEqual,
		Xor,
		BitwiseAnd,
		BitwiseOr,
		ShiftRight,
		ShiftLeft,
		LParen,
		RParen,
		LCurlyParen,
		RCurlyParen,
		LSquareParen,
		RSquareParen,
		Function, //
		Procedure, //
		AnonFunc, //
		AnonProc, //
		Struct, //
		ReturnType, //
		As, //
		Takes, //
		Default,
		Return, //
		//EndProc, //
		//EndStruct, //
		If, //
		Else,
		Do,
		For, //
		Of,
		While, //
		//EndIf, //
		//EndFor, //
		//EndWhile, //
		End,
		Input, //
		Output, //
		Create, //
		DictEquator,
		Cast, //
		Call,
		With,
		Raw, //kinda - not actually used as a token, just consumes the rest of the line to creat
e a string literal
		Quit,
		Require,
		Set, //
		To,
		Type, //
		Identifier,
```

```
86:                 SingleLineComment,
87:                 StartMultiLineComment,
88:                 EndMultiLineComment,
89:                 Whitespace,
90:                 Newline,
91:                 Comma,
92:                 UnfinishedToken,
93:                 ReturnValue, //Possibly unused
94:                 BooleanExpr,
95:                 Break,
96:                 eof,
97:                 VarAccessOp,
98:                 Import,
99:                 NoImport,
100:                NoInstall,
101:                NoBare
102:        };
103:
104:        enum class VarType
105:        {
106:                Int,
107:                Float,
108:                Complex,
109:                Bool,
110:                String,
111:                List,
112:                Dict,
113:                Function,
114:                None
115:        };
116:
117:        inline std::string TokenTypeName(TokenType type)
118:        {
119:                switch (type)
120:                {
121:                case TokenType::None:             return "None";
122:                case TokenType::Int:              return "Int";
123:                case TokenType::Float:            return "Float";
124:                case TokenType::Complex:          return "Complex";
125:                case TokenType::PositiveTypeMod:  return "PositiveTypeMod";
126:                case TokenType::NegativeTypeMod:  return "NegativeTypeMod";
127:                case TokenType::NonZeroTypeMod:   return "NonZeroTypeMod";
128:                case TokenType::Increment:                return "Increment";
129:                case TokenType::Decrement:                return "Decrement";
130:                case TokenType::String:           return "String";
131:                case TokenType::True:             return "True";
132:                case TokenType::False:            return "False";
133:                case TokenType::Type:             return "Type";
134:                case TokenType::Plus:             return "Plus";
135:                case TokenType::Minus:            return "Minus";
136:                case TokenType::Multiply:         return "Multiply";
137:                case TokenType::Divide:           return "Divide";
138:                case TokenType::Intdiv:           return "Intdiv";
139:                case TokenType::Modulo:           return "Modulo";
140:                case TokenType::Power:            return "Power";
141:                case TokenType::Is:               return "Is";
142:                case TokenType::And:              return "And";
143:                case TokenType::Or:               return "Or";
144:                case TokenType::Not:              return "Not";
145:                case TokenType::Equal:            return "Equal";
146:                case TokenType::NotEqual:         return "NotEqual";
147:                case TokenType::LessThan:         return "LessThan";
148:                case TokenType::GreaterThan:      return "GreaterThan";
149:                case TokenType::LessThanEqual:    return "LessThanEqual";
150:                case TokenType::GreaterThanEqual: return "GreaterThanEqual";
151:                case TokenType::Xor:              return "Xor";
152:                case TokenType::BitwiseAnd:       return "BitwiseAnd";
153:                case TokenType::BitwiseOr:        return "BitwiseOr";
154:                case TokenType::ShiftRight:       return "ShiftRight";
155:                case TokenType::ShiftLeft:        return "ShiftLeft";
156:                case TokenType::LParen:           return "LParen";
157:                case TokenType::RParen:           return "RParen";
158:                case TokenType::Function:         return "Function";
159:                case TokenType::Procedure:        return "Procedure";
160:                case TokenType::AnonFunc:         return "AnonFunc";
161:                case TokenType::AnonProc:         return "AnonProc";
162:                case TokenType::Struct:           return "Struct";
163:                case TokenType::ReturnType:       return "ReturnType";
164:                case TokenType::Takes:            return "Takes";
165:                case TokenType::As:               return "As";
166:                case TokenType::Return:           return "Return";
167:                case TokenType::End:              return "End";
168:                //case TokenType::EndStruct:        return "EndStruct";
169:                case TokenType::If:               return "If";
170:                case TokenType::For:              return "For";
171:                case TokenType::While:            return "While";
```

```
  172:                /*case TokenType::EndIf:           return "EndIf";
  173:                case TokenType::EndFor:          return "EndFor";
  174:                case TokenType::EndWhile:        return "EndWhile";*/
  175:                case TokenType::Input:           return "Input";
  176:                case TokenType::Output:          return "Output";
  177:                case TokenType::Create:          return "Create";
  178:                case TokenType::Cast:            return "Cast";
  179:                case TokenType::Call:            return "Call";
  180:                case TokenType::With:            return "With";
  181:                case TokenType::Raw:             return "Raw";
  182:                case TokenType::Quit:            return "Quit";
  183:                case TokenType::Require:         return "Require";
  184:                case TokenType::Set:             return "Set";
  185:                case TokenType::To:              return "To";
  186:                //case TokenType::Type:          return "Type";
  187:                case TokenType::Identifier:      return "Identifier";
  188:                case TokenType::UnfinishedToken: return "UnfinishedToken";
  189:                case TokenType::ReturnValue:     return "ReturnValue";
  190:                case TokenType::BooleanExpr:     return "BooleanExpr";
  191:                case TokenType::Comma:           return "Comma";
  192:                case TokenType::Whitespace:      return "Whitespace";
  193:                case TokenType::Newline:         return "Newline";
  194:                case TokenType::eof:             return "EOF";
  195:                case TokenType::VarAccessOp:     return "VarAccessOperator";
  196:                default:                         return "Unknown";
  197:                }
  198:        }
  199:
  200:        class Token
  201:        {
  202:        public:
  203:                Token(TokenType tokenType, const char* contents, uint32_t lineNumber, uint32_t character
Number)
  204:                        : m_Type(tokenType), m_Contents(contents), m_LineNumber(lineNumber), m_Character
Number(characterNumber)
  205:                {  }
  206:
  207:                virtual ~Token() = default;
  208:
  209:                inline TokenType GetType() { return m_Type; }
  210:                inline std::string GetContents() { return m_Contents; }
  211:                inline uint32_t GetLineNumber() { return m_LineNumber; }
  212:                inline uint32_t GetCharacterNumber() { return m_CharacterNumber; }
  213:                inline void AppendContents(std::string a) { m_Contents += a; }
  214:
  215:                inline std::string ToString() const { return "[Token " + std::to_string(m_LineNumber) +
"," + std::to_string(m_CharacterNumber) + ": " + TokenTypeName(m_Type) + ": " + m_Contents + "]"; }
  216:        private:
  217:                TokenType m_Type;
  218:                std::string m_Contents;
  219:                uint32_t m_LineNumber;
  220:                uint32_t m_CharacterNumber;
  221:        };
  222:
  223:        inline std::ostream& operator<<(std::ostream& os, const Token& t)
  224:        {
  225:                // [Token: Int: 0]
  226:                os << t.ToString();
  227:                return os;
  228:        }
  229: }
```

```
    1: #pragma once
    2: #include <string>
    3: #include "Log.h"
    4:
    5: namespace Spliwaca
    6: {
    7:         class LexicalError
    8:         {
    9:         public:
   10:                 LexicalError(uint8_t errorCode, uint32_t lineNumber, uint32_t columnNumber, uint16_t col
umnSpan = 1)
   11:                         : m_ErrorCode(errorCode), m_LineNumber(lineNumber), m_ColumnNumber(columnNumber)
, m_ColumnSpan(columnSpan)
   12:                 {
   13:                 }
   14:                 ~LexicalError() = default;
   15:
   16:                 inline const uint8_t  GetErrorCode() const { return m_ErrorCode; }
   17:                 inline const uint32_t GetLineNumber() const { return m_LineNumber; }
   18:                 inline const uint32_t GetColumnNumber() const { return m_ColumnNumber; }
   19:                 inline const uint16_t GetColumnSpan() const { return m_ColumnSpan; }
   20:
   21:         private:
   22:                 uint8_t m_ErrorCode;
   23:                 uint32_t m_LineNumber;
   24:                 uint32_t m_ColumnNumber;
   25:                 uint16_t m_ColumnSpan;
   26:         };
   27: };
```

```
    1: #pragma once
    2: #include <cstdint>
    3:
    4: namespace Spliwaca
    5: {
    6:         enum class SyntaxErrorType : uint8_t
    7:         {
    8:                 expNewline = 0,   // 0
    9:                 expIdent,         // 1
   10:                 expStatement,     // 2
   11:                 expAtom,          // 3
   12:                 expType,          // 4
   13:                 expComma,         // 5
   14:                 expRParen,        // 6
   15:                 expRSquareParen,  // 7
   16:                 expDo,            // 8
   17:                 expTo,            // 9
   18:                 expOf,            // 10
   19:                 expWith,          // 11
   20:                 expTakes,         // 12
   21:                 expReturns,       // 13
   22:                 expAs,            // 14
   23:                 expRaw,           // 15
   24:                 expEndIf,         // 16
   25:                 expEndFor,        // 17
   26:                 expEndWhile,      // 18
   27:                 expEndFunc,           // 19
   28:                 expEndProc,           // 20
   29:                 expEndStruct,     // 21
   30:                 expTypeMod,           // 22
   31:                 tooManyElse,      // 23
   32:                 unexpEndFunc,     // 24
   33:                 unexpEndProc,     // 25
   34:                 unexpEndIf,           // 26
   35:                 unexpEndFor,      // 27
   36:                 unexpEndWhile,    // 28
   37:                 unexpEndStruct,   // 29
   38:                 unexpElseIf,      // 30
   39:                 inconsistentDict  // 31
   40:         };
   41:
   42:         class SyntaxError
   43:         {
   44:         public:
   45:                 SyntaxError(SyntaxErrorType errorCode, std::shared_ptr<Token> token)
   46:                         : m_ErrorCode(errorCode), m_LineNumber(token->GetLineNumber()), m_ColumnNumber(t
oken->GetCharacterNumber()),
   47:                           m_ColumnSpan(token->GetContents().length()), m_TokenType(token->GetType())
   48:                 {}
   49:
   50:                 SyntaxError(SyntaxErrorType errorCode, uint32_t lineNumber, uint32_t columnNumber, size_
t columnSpan, Spliwaca::TokenType type)
   51:                         : m_ErrorCode(errorCode), m_LineNumber(lineNumber), m_ColumnNumber(columnNumber)
,
   52:                         m_ColumnSpan(columnSpan), m_TokenType(type)
   53:                 {}
   54:
   55:                 ~SyntaxError() = default;
   56:
   57:                 inline const SyntaxErrorType  GetErrorCode() const { return m_ErrorCode; }
   58:                 inline const uint32_t GetLineNumber() const { return m_LineNumber; }
   59:                 inline const uint32_t GetColumnNumber() const { return m_ColumnNumber; }
   60:                 inline const size_t GetColumnSpan() const { return m_ColumnSpan; }
   61:                 inline const TokenType GetTokenType() const { return m_TokenType; }
   62:
   63:                 //2^8 = 256 error codes available
   64:                 //Code 0: Unexpected Token -- expected newline
   65:                 //Code 1: Unexpected Token -- expected identifier
   66:                 //Code 2: Unexpected Token -- expected statement
   67:                 //Code 3: Unexpected Token -- expected do
   68:                 //Code 4: Unexpected Token -- expected "END IF" token pair
   69:                 //Code 5: Unexpected Token "ELSE" -- cannot have more than one else in an if tree
   70:
   71:         private:
   72:                 SyntaxErrorType m_ErrorCode;
   73:                 uint32_t m_LineNumber, m_ColumnNumber;
   74:                 size_t m_ColumnSpan;
   75:                 TokenType m_TokenType;
   76:                 //std::shared_ptr<Token> m_OffendingToken;
   77:         };
   78: }
```

```
    1: #pragma once
    2: #include <memory>
    3: #include <vector>
    4: #include "Frontend/Lexer/Token.h"
    5: #include "Nodes.h"
    6: #include "Frontend/Scopes/Scope.h"
    7:
    8: namespace Spliwaca
    9: {
   10:         class Parser
   11:         {
   12:         public:
   13:                 static std::shared_ptr<Parser> Create(std::shared_ptr<std::vector<std::shared_ptr<Token>
>> tokens);
   14:                 ~Parser() = default;
   15:
   16:                 std::shared_ptr<EntryPoint> ConstructAST();
   17:
   18:         private:
   19:                 inline uint32_t IncIndex() { m_TokenIndex++; return m_TokenIndex; }
   20:
   21:                 Parser(std::shared_ptr<std::vector<std::shared_ptr<Token>>> tokens)
   22:                         : m_Tokens(tokens), m_TokenIndex(0) { }
   23:
   24:                 std::shared_ptr<std::vector<std::shared_ptr<Token>>> m_Tokens;
   25:                 uint32_t m_TokenIndex;
   26:
   27:                 //std::shared_ptr<Scope> m_MainScope;
   28:                 //std::vector<std::shared_ptr<Scope>> m_ScopeStack;
   29:                 //std::shared_ptr<Scope> m_CurrentScope;
   30:
   31:         private:
   32:                 std::shared_ptr<RequireNode> ConstructRequire();
   33:                 std::shared_ptr<Statements> ConstructStatements();
   34:                 std::shared_ptr<Statement> ConstructStatement();
   35:
   36:                 std::shared_ptr<IfNode> ConstructIf();
   37:                 std::shared_ptr<SetNode> ConstructSet();
   38:                 std::shared_ptr<InputNode> ConstructInput();
   39:                 std::shared_ptr<OutputNode> ConstructOutput();
   40:                 std::shared_ptr<IncNode> ConstructIncrement();
   41:                 std::shared_ptr<DecNode> ConstructDecrement();
   42:                 std::shared_ptr<ForNode> ConstructFor();
   43:                 std::shared_ptr<WhileNode> ConstructWhile();
   44:                 std::shared_ptr<QuitNode> ConstructQuit();
   45:                 std::shared_ptr<CallNode> ConstructCall();
   46:                 std::shared_ptr<ImportNode> ConstructImport();
   47:                 std::shared_ptr<FuncNode> ConstructFunction();
   48:                 std::shared_ptr<ProcNode> ConstructProcedure();
   49:                 std::shared_ptr<StructNode> ConstructStruct();
   50:
   51:                 std::shared_ptr<Expr> ConstructExpr();
   52:
   53:                 std::shared_ptr<ListNode> ConstructList();
   54:                 std::shared_ptr<DictEntryNode> ConstructDictEntry();
   55:                 /*std::shared_ptr<BoolExprNode> ConstructBooleanExpr();
   56:                 std::shared_ptr<AddExprNode> ConstructAddExpr();
   57:                 std::shared_ptr<MulExprNode> ConstructMulExpr();
   58:                 std::shared_ptr<DivModExprNode> ConstructDivModExpr();
   59:                 std::shared_ptr<PowerNode> ConstructPower();*/
   60:                 std::shared_ptr<BinOpNode> ConstructBinOpNode();
   61:                 std::shared_ptr<FactorNode> ConstructFactor();
   62:                 std::shared_ptr<AtomNode> ConstructAtom(bool quit = false);
   63:                 std::shared_ptr<ListAccessNode> ConstructListAccess();
   64:
   65:                 std::shared_ptr<CreateNode> ConstructCreate();
   66:                 std::shared_ptr<CastNode> ConstructCast();
   67:                 std::shared_ptr<ReturnNode> ConstructReturn();
   68:                 std::shared_ptr<AnonfNode> ConstructAnonFunc();
   69:                 std::shared_ptr<AnonpNode> ConstructAnonProc();
   70:
   71:                 std::shared_ptr<TypeNode> ConstructTypeNode();
   72:                 std::shared_ptr<IdentNode> ConstructIdentNode();
   73:
   74:         };
   75: }
```

```
    1: #pragma once
    2: #include <memory>
    3: #include <vector>
    4: #include <string>
    5: #include "Frontend/Lexer/Token.h"
    6: #include <Log.h>
    7: #include <map>
    8:
    9: namespace Spliwaca
   10: {
   11:         struct Statements;
   12:         struct Expr;
   13:         struct AtomNode;
   14:         struct CallNode;
   15:         struct ListNode;
   16:         //class BoolExprNode;
   17:         //class MulExprNode;
   18:
   19:         class ImportConfig {
   20:         public:
   21:                 ImportConfig(bool allowImport, bool allowPyImport, bool allowInstall, bool allowBare)
   22:                         : allowImport(allowImport), allowPyImport(allowPyImport), allowInstall(allowInst
all), allowBare(allowBare) {}
   23:                 bool allowImport;
   24:                 bool allowPyImport;
   25:                 bool allowInstall;
   26:                 bool allowBare;
   27:         };
   28:
   29:         class IdentNode
   30:         {
   31:         public:
   32:
   33:                 std::vector<std::shared_ptr<Token>> ids;
   34:                 bool accessPresent = false;
   35:
   36:                 std::string GetContents();
   37:                 std::string GenerateGetattrTree(ImportConfig *importConfig, bool &interpreter_var, bool
minus_one = false);
   38:                 std::string GenerateGetattrTree(ImportConfig *importConfig, bool minus_one = false);
   39:                 std::string GetFinalId();
   40:
   41:                 inline uint32_t GetLineNumber() { return ids.at(0)->GetLineNumber(); }
   42:                 inline uint32_t GetColumnNumber() { return ids.at(0)->GetCharacterNumber(); }
   43:                 inline uint32_t GetIdentAccessNum() { return ids.size(); }
   44:
   45:                 IdentNode()
   46:                 {
   47:                 }
   48:
   49:         private:
   50:                 std::string cachedContents = "";
   51:                 std::string cachedGetattrMinusOne = "";
   52:                 std::string cachedGetattr = "";
   53:
   54:         };
   55:
   56:         struct TypeNode
   57:         {
   58:                 std::shared_ptr<IdentNode> ident; // 1
   59:                 std::shared_ptr<Token> typeToken; // 2
   60:                 int type;
   61:         };
   62:
   63:         struct AnonpNode
   64:         {
   65:                 std::vector<std::shared_ptr<TypeNode>> argTypes;
   66:                 std::vector<std::shared_ptr<IdentNode>> argNames;
   67:                 std::shared_ptr<Statements> body;
   68:         };
   69:
   70:         struct AnonfNode
   71:         {
   72:                 std::vector<std::shared_ptr<TypeNode>> argTypes;
   73:                 std::vector<std::shared_ptr<IdentNode>> argNames;
   74:                 std::shared_ptr<TypeNode> returnType;
   75:                 std::shared_ptr<Statements> body;
   76:         };
   77:
   78:         struct CastNode
   79:         {
   80:                 std::shared_ptr<TypeNode> castType;
   81:                 std::shared_ptr<ListNode> list;
   82:         };
   83:
   84:         struct CreateNode
```

```
85:         {
86:                 std::shared_ptr<TypeNode> createType;
87:                 std::vector<std::shared_ptr<Expr>> args;
88:         };
89:
90:         struct ListAccessNode
91:         {
92:                 std::vector<std::shared_ptr<ListNode>> indices;
93:         };
94:
95:         struct AtomNode
96:         {
97:                 std::shared_ptr<Token> token; //type: 1
98:                 std::shared_ptr<ListNode> list; //type: 2
99:                 std::shared_ptr<IdentNode> ident; //type: 3
100:                std::shared_ptr<ListAccessNode> listAccess;
101:                uint8_t type;
102:                bool listAccessPresent;
103:        };
104:
105:        struct FactorNode
106:        {
107:                std::shared_ptr<Token> opToken;
108:                std::shared_ptr<AtomNode> right;
109:                bool opTokenPresent;
110:        };
111:
112:        /*
113:        struct PowerNode
114:        {
115:                std::shared_ptr<FactorNode> left;
116:                std::shared_ptr<Token> opToken;
117:                std::shared_ptr<PowerNode> right;
118:        };
119:
120:        struct DivModExprNode
121:        {
122:                std::shared_ptr<PowerNode> left;
123:                std::shared_ptr<Token> opToken;
124:                std::shared_ptr<DivModExprNode> right;
125:        };
126:
127:        class MulExprNode
128:        {
129:        public:
130:
131:                std::shared_ptr<DivModExprNode> left;
132:                std::shared_ptr<Token> opToken;
133:                std::shared_ptr<MulExprNode> right;
134:        };
135:
136:        class AddExprNode
137:        {
138:        public:
139:                VarType GetExprReturnType();
140:
141:                AddExprNode()
142:                {
143:                }
144:
145:                std::shared_ptr<MulExprNode> left;
146:                std::shared_ptr<Token> opToken;
147:                std::shared_ptr<AddExprNode> right;
148:        };
149:
150:        class BoolExprNode
151:        {
152:        public:
153:                VarType GetExprReturnType()
154:                {
155:                        if (opToken != nullptr)
156:                        {
157:                                return VarType::Bool;
158:                        }
159:                        else
160:                        {
161:                                return left->GetExprReturnType();
162:                        }
163:                }
164:
165:                BoolExprNode()
166:                {
167:                }
168:
169:                std::shared_ptr<AddExprNode> left;
170:                std::shared_ptr<Token> opToken;
```

```
171:                std::shared_ptr<BoolExprNode> right;
172:                int exprType;
173:        };
174:        */
175:
176:        struct BinOpNode
177:        {
178:                std::shared_ptr<FactorNode> left;
179:                std::shared_ptr<Token> opToken;
180:                std::shared_ptr<BinOpNode> right;
181:        };
182:
183:        struct Expr
184:        {
185:                std::shared_ptr<BinOpNode> binOpNode; // exprType: 1
186:                std::shared_ptr<CreateNode> createNode; // exprType: 2
187:                std::shared_ptr<CastNode> castNode; // exprType: 3
188:                std::shared_ptr<CallNode> callNode; // exprType: 4
189:                std::shared_ptr<AnonfNode> anonfNode; // exprType: 5
190:                std::shared_ptr<AnonpNode> anonpNode; // exprType: 6
191:                uint8_t exprType;
192:        };
193:
194:        struct DictEntryNode
195:        {
196:                std::shared_ptr<Expr> left;
197:                std::shared_ptr<Expr> right;
198:                bool hasRight;
199:        };
200:
201:        struct ListNode
202:        {
203:                std::vector<std::shared_ptr<DictEntryNode>> Items;
204:        };
205:
206:        struct ProcNode
207:        {
208:                std::shared_ptr<IdentNode> id;
209:                std::vector<std::shared_ptr<TypeNode>> argTypes;
210:                std::vector<std::shared_ptr<IdentNode>> argNames;
211:                std::shared_ptr<Statements> body;
212:                uint32_t lineNumber;
213:        };
214:
215:        struct FuncNode
216:        {
217:                std::shared_ptr<IdentNode> id;
218:                std::vector<std::shared_ptr<TypeNode>> argTypes;
219:                std::vector<std::shared_ptr<IdentNode>> argNames;
220:                std::shared_ptr<TypeNode> returnType;
221:                std::shared_ptr<Statements> body;
222:                uint32_t lineNumber;
223:        };
224:
225:        struct StructNode
226:        {
227:                std::shared_ptr<IdentNode> id;
228:                std::vector<std::shared_ptr<TypeNode>> types;
229:                std::vector<std::shared_ptr<IdentNode>> names;
230:                uint32_t lineNumber;
231:        };
232:
233:        struct ImportNode
234:        {
235:                std::shared_ptr<IdentNode> id;
236:        };
237:
238:        struct ReturnNode
239:        {
240:                std::shared_ptr<ListNode> list;
241:        };
242:
243:        struct CallNode
244:        {
245:                std::shared_ptr<Expr> function;
246:                std::vector<std::shared_ptr<Expr>> args;
247:        };
248:
249:        struct QuitNode
250:        {
251:                std::shared_ptr<AtomNode> returnVal;
252:        };
253:
254:        struct WhileNode
255:        {
256:                std::shared_ptr<BinOpNode> condition;
```

```
257:                std::shared_ptr<Statements> body;
258:                uint32_t lineNumber;
259:        };
260:
261:        struct ForNode
262:        {
263:                std::shared_ptr<IdentNode> id;
264:                std::shared_ptr<ListNode> iterableExpr;
265:                std::shared_ptr<Statements> body;
266:                uint32_t lineNumber;
267:        };
268:
269:        struct DecNode
270:        {
271:                std::shared_ptr<IdentNode> id;
272:        };
273:
274:        struct IncNode
275:        {
276:                std::shared_ptr<IdentNode> id;
277:        };
278:
279:        struct OutputNode
280:        {
281:                std::shared_ptr<Token> raw;
282:        };
283:
284:        struct InputNode
285:        {
286:                std::shared_ptr<Token> signSpec;
287:                std::shared_ptr<TypeNode> type;
288:                std::shared_ptr<IdentNode> id;
289:        };
290:
291:        struct SetNode
292:        {
293:                std::shared_ptr<IdentNode> id;
294:                std::shared_ptr<ListNode> list;
295:        };
296:
297:        struct IfNode
298:        {
299:                std::vector<std::shared_ptr<ListNode>> conditions;
300:                std::vector<std::shared_ptr<Statements>> bodies;
301:                bool elsePresent;
302:                std::vector<uint32_t> lineNumbers;
303:        };
304:
305:        struct Statement
306:        {
307:                std::shared_ptr<IfNode> ifNode;
308:                std::shared_ptr<SetNode> setNode;
309:                std::shared_ptr<InputNode> inputNode;
310:                std::shared_ptr<OutputNode> outputNode;
311:                std::shared_ptr<IncNode> incNode;
312:                std::shared_ptr<DecNode> decNode;
313:                std::shared_ptr<ForNode> forNode;
314:                std::shared_ptr<WhileNode> whileNode;
315:                std::shared_ptr<QuitNode> quitNode;
316:                std::shared_ptr<CallNode> callNode;
317:                std::shared_ptr<FuncNode> funcNode;
318:                std::shared_ptr<ProcNode> procNode;
319:                std::shared_ptr<StructNode> structNode;
320:                std::shared_ptr<ReturnNode> returnNode;
321:                std::shared_ptr<ImportNode> importNode;
322:                uint8_t statementType;
323:                uint32_t lineNumber;
324:        };
325:
326:        struct Statements
327:        {
328:                std::vector<std::shared_ptr<Statement>> statements;
329:        };
330:
331:        struct RequireNode
332:        {
333:                std::shared_ptr<IdentNode> requireType;
334:        };
335:
336:        struct VarInfo {
337:                int type; // 0: Callable, 1: Variable
338:                int declLine;
339:                union {
340:                        char flags;
341:                        struct {
342:                                char UndefinedAtFirstUse : 1;
```

```
343:                                char GlobalRead : 1;
344:                                char GlobalMod : 1;
345:
346:                        } Flags;
347:                };
348:        };
349:
350:        struct SemanticState {
351:                std::string currentScope = "globals";
352:                std::map<std::string, VarInfo> variables = {};
353:                std::string setVar = "";
354:                bool set = false;
355:                bool incdec = false;
356:        };
357:
358:        struct EntryPoint
359:        {
360:                std::shared_ptr<RequireNode> require;
361:                std::shared_ptr<Statements> statements;
362:                bool requirePresent;
363:
364:                SemanticState *semanticState;
365:        };
366: }
```

```
 1: #pragma once
 2:
 3: namespace Spliwaca
 4: {
 5:         enum class SemanticErrorType
 6:         {
 7:                 MissingVariable,
 8:                 StringOperationTypeMismatch
 9:         };
10: }
```

```
    1: #pragma once
    2: #include <vector>
    3: #include <map>
    4: #include <string>
    5: #include <functional>
    6: #include <Log.h>
    7:
    8: namespace Spliwaca
    9: {
   10:         enum class ScopeType
   11:         {
   12:                 Main = 0,
   13:                 If,
   14:                 For,
   15:                 While,
   16:                 Function,
   17:                 Procedure,
   18:                 Anonf,
   19:                 Anonp,
   20:                 Struct
   21:         };
   22:
   23:         class ScopeEntry
   24:         {
   25:         public:
   26:                 ScopeEntry(std::string symbol, int lineNumber, VarType type)
   27:                         : m_Symbol(symbol), m_LineNumber(lineNumber), m_AmbiguousType(false), m_SymbolTy
pe(type)
   28:                 {
   29:                 }
   30:
   31:                 ScopeEntry()
   32:                         : m_Symbol(""), m_LineNumber(0), m_AmbiguousType(false), m_SymbolType(VarType::N
one) { }
   33:
   34:                 const std::string GetSymbol() const { return m_Symbol; }
   35:                 const int GetLineNumber() const { return m_LineNumber; }
   36:                 const VarType GetSymbolType() const { return m_SymbolType; }
   37:
   38:                 void Ambiguous() { m_AmbiguousType = true; }
   39:         private:
   40:                 std::string m_Symbol;
   41:                 uint32_t m_LineNumber;
   42:                 VarType m_SymbolType;
   43:                 bool m_AmbiguousType;
   44:         };
   45:
   46:         class Scope
   47:         {
   48:         public:
   49:                 Scope(std::string name, uint32_t lineNumber, ScopeType type)
   50:                         : m_Name(name), m_StartLineNumber(lineNumber), m_Type(type)
   51:                 {
   52:                         if (m_Type == ScopeType::Anonf || m_Type == ScopeType::Anonp || m_Type == ScopeT
ype::Function || m_Type == ScopeType::Procedure)
   53:                                 m_BlockingScope = true;
   54:                         else
   55:                                 m_BlockingScope = false;
   56:                 }
   57:
   58:                 void CloseScope(uint32_t lineNumber)
   59:                 {
   60:                         m_EndLineNumber = lineNumber;
   61:                         m_Readonly = true;
   62:                 }
   63:
   64:                 void AddEntry(std::string symbol, uint32_t lineNumber, VarType type)
   65:                 {
   66:                         if (m_Readonly)
   67:                         {
   68:                                 SPLW_WARN("Attempted to add an entry to a closed scope. This should not
happen.");
   69:                                 return;
   70:                         }
   71:                         size_t hashed = std::hash<std::string>()(symbol);
   72:                         m_Entries.insert(std::map<size_t, std::shared_ptr<ScopeEntry>>::value_type(hashe
d, std::make_shared<ScopeEntry>(symbol, lineNumber, type)));
   73:                 }
   74:
   75:                 std::shared_ptr<Scope> AddSubScope(std::string name, uint32_t lineNumber, ScopeType type
)
   76:                 {
   77:                         if (m_Readonly)
   78:                         {
   79:                                 SPLW_WARN("Attempted to add a sub-scope to a closed scope. This should n
ot happen.");
```

```
80:                                 return nullptr;
81:                         }
82:                         m_SubScopes.push_back(std::make_shared<Scope>(name, lineNumber, type));
83:                         return m_SubScopes.at(m_SubScopes.size() - 1);
84:                 }
85:
86:                 std::shared_ptr<ScopeEntry> FindIdent(std::shared_ptr<IdentNode> ident)
87:                 {
88:                         size_t hashed = std::hash<std::string>()(ident->GetContents());
89:                         if (m_Entries.find(hashed) != m_Entries.end())
90:                                 return m_Entries[hashed];
91:                         else if (ident->GetIdentAccessNum() == 1)
92:                                 return nullptr;
93:                         return nullptr;
94:                 }
95:
96:                 ScopeType GetType() { return m_Type; }
97:
98:         private:
99:                 std::map<size_t, std::shared_ptr<ScopeEntry>> m_Entries;
100:                std::vector<std::shared_ptr<Scope>> m_SubScopes;
101:                std::shared_ptr<Scope> m_ParentScope;
102:
103:                std::string m_Name;
104:                ScopeType m_Type;
105:                uint32_t m_StartLineNumber;
106:                uint32_t m_EndLineNumber;
107:                bool m_Readonly = false;
108:                bool m_BlockingScope;
109:        };
110:
111:        class StructScope : Scope
112:        {
113:        public:
114:                StructScope(std::string name, uint32_t lineNumber, ScopeType type)
115:                        : Scope(name, lineNumber, type) {}
116:
117:        private:
118:                bool m_Accessible = false;
119:        };
120:
121:        class MainScope : Scope
122:        {
123:        public:
124:                MainScope(std::string name, uint32_t lineNumber, ScopeType type)
125:                        : Scope(name, lineNumber, type) {}
126:
127:        private:
128:                std::map<size_t, StructScope> m_Structs;
129:        };
130: }
```

```
#pragma once
#include "Frontend/Parser/Nodes.h"

namespace Spliwaca
{
	class Generator
	{
	public:
		static std::shared_ptr<Generator> Create(std::shared_ptr<EntryPoint> entryPoint);
		~Generator() = default;

		std::string GenerateCode(int &errorCode);

		Generator(std::shared_ptr<EntryPoint> ep)
			: m_EntryPoint(ep), m_Tabs(""), m_Code("")
		{}

	private:
		std::shared_ptr<EntryPoint> m_EntryPoint;
		std::string m_Tabs;
		std::string m_Code;
		std::string m_CurrentFuncNameLine;

		bool m_AbortPrint = false;
		bool m_InterpreterCall = false;

		std::vector<ImportConfig*> m_ScopeImportConfigs;

	private:
		ImportConfig *getCurrentImportConfig() const { return m_ScopeImportConfigs.at(m_ScopeImp
ortConfigs.size() - 1); };
		void GenerateStatements(std::shared_ptr<Statements> s);

		void GenerateIf(std::shared_ptr<IfNode> node);
		void GenerateSet(std::shared_ptr<SetNode> node);
		void GenerateInput(std::shared_ptr<InputNode> node);
		void GenerateOutput(std::shared_ptr<OutputNode> node);
		void GenerateInc(std::shared_ptr<IncNode> node);
		void GenerateDec(std::shared_ptr<DecNode> node);
		void GenerateFor(std::shared_ptr<ForNode> node);
		void GenerateWhile(std::shared_ptr<WhileNode> node);
		void GenerateQuit(std::shared_ptr<QuitNode> node);
		void GenerateCall(std::shared_ptr<CallNode> node, bool statement);
		void GenerateFunc(std::shared_ptr<FuncNode> node);
		void GenerateProc(std::shared_ptr<ProcNode> node);
		void GenerateStruct(std::shared_ptr<StructNode> node);
		void GenerateReturn(std::shared_ptr<ReturnNode> node);
		void GenerateImport(std::shared_ptr<ImportNode> node);

		void GenerateList(std::shared_ptr<ListNode> node, bool fromAtom = false);
		void GenerateDictEntry(std::shared_ptr<DictEntryNode> node);
		void GenerateExpr(std::shared_ptr<Expr> node);
		void GenerateBinOp(std::shared_ptr<BinOpNode> node);
		void GenerateFactor(std::shared_ptr<FactorNode> node);
		void GenerateAtom(std::shared_ptr<AtomNode> node);

		void GenerateCreate(std::shared_ptr<CreateNode> node);
		void GenerateCast(std::shared_ptr<CastNode> node);
		void GenerateAnonf(std::shared_ptr<AnonfNode> node);
		void GenerateAnonp(std::shared_ptr<AnonpNode> node);

		void GenerateType(std::shared_ptr<TypeNode> node);

		std::string ParseRaw(std::shared_ptr<Token> token);
		std::string ParseComplex(std::shared_ptr<Token> token);
		std::string StripLeadingZeros(std::string token);
	};

	/*std::shared_ptr<IfNode> ifNode;
		std::shared_ptr<SetNode> setNode;
		std::shared_ptr<InputNode> inputNode;
		std::shared_ptr<OutputNode> outputNode;
		std::shared_ptr<IncNode> incNode;
		std::shared_ptr<DecNode> decNode;
		std::shared_ptr<ForNode> forNode;
		std::shared_ptr<WhileNode> whileNode;
		std::shared_ptr<QuitNode> quitNode;
		std::shared_ptr<CallNode> callNode;
		std::shared_ptr<FuncNode> funcNode;
		std::shared_ptr<ProcNode> procNode;
		std::shared_ptr<StructNode> structNode;
		std::shared_ptr<ReturnNode> returnNode;*/
}
```

```
    1: #pragma once
    2: #include "Frontend/Lexer/Lexer.h"
    3: #include "Frontend/Lexer/LexicalError.h"
    4: #include "Frontend/Parser/Parser.h"
    5: #include "Frontend/Parser/SyntaxError.h"
    6: #include "Backend/Code Generation/Generator.h"
    7: #include "Frontend/Parser/SemanticError.h"
    8: #include "UtilFunctions.h"
    9:
   10:
   11: namespace Spliwaca
   12: {
   13:         struct TranspilerState
   14:         {
   15:                 std::vector<LexicalError> LexerErrors;
   16:                 std::vector<SyntaxError> SyntaxErrors;
   17:                 //std::vector<MissingVariable> MissingVariables;
   18:         };
   19:
   20:         class Transpiler
   21:         {
   22:         public:
   23:                 Transpiler(std::string filename, std::string output, std::shared_ptr<TranspilerState> st
ate, bool printTokenList)
   24:                         : m_Filename(filename), m_Output(output), m_State(state), m_PrintTokenList(print
TokenList)
   25:                 {
   26:                 }
   27:
   28:                 std::string Run();
   29:
   30:         private:
   31:                 std::string GetSyntaxErrorMessage(SyntaxErrorType type)
   32:                 {
   33:                         switch (type)
   34:                         {
   35:                         case SyntaxErrorType::expNewline:   // 0
   36:                                 return "Expected newline, got {0}";
   37:                         case SyntaxErrorType::expIdent:         // 1
   38:                                 return "Expected identifier, got {0}";
   39:                         case SyntaxErrorType::expStatement:     // 2
   40:                                 return "Expected statement beginning, got incompatible token type {0}";
   41:                         case SyntaxErrorType::expAtom:          // 3
   42:                                 return "Expected atom (value/list), got incompatible token type {0}";
   43:                         case SyntaxErrorType::expType:          // 4
   44:                                 return "Expected type, got {0}";
   45:                         case SyntaxErrorType::expComma:         // 5
   46:                                 return "Expected a comma, got {0}";
   47:                         case SyntaxErrorType::expRParen:        // 6
   48:                                 return "Expected a right parenthesis, got {0}";
   49:                         case SyntaxErrorType::expRSquareParen: // 7
   50:                                 return "Expected right square parenthesis, got {0}";
   51:                         case SyntaxErrorType::expDo:         // 8
   52:                                 return "Expected DO, got {0}";
   53:                         case SyntaxErrorType::expTo:              // 9
   54:                                 return "Expected TO, got {0}";
   55:                         case SyntaxErrorType::expOf:              // 10
   56:                                 return "Expected OF, got {0}";
   57:                         case SyntaxErrorType::expWith:          // 11
   58:                                 return "Expected WITH, got {0}";
   59:                         case SyntaxErrorType::expTakes:        // 12
   60:                                 return "Expected TAKES, got {0}. This error shouldn't happen, since TAKE
S is optional.";
   61:                         case SyntaxErrorType::expReturns:       // 13
   62:                                 return "Expected RETURNS, got {0}. Functions must specify a return type
and return a value.";
   63:                         case SyntaxErrorType::expAs:              // 14
   64:                                 return "Expected AS, got {0}";
   65:                         case SyntaxErrorType::expRaw:          // 15
   66:                                 return "Expected a raw token i.e. a string. This error should not occur.
 Got {0}";
   67:                         case SyntaxErrorType::expEndIf:         // 16
   68:                                 return "Expected END IF, got {0}. Reached the end of a statement block,
but there was an incorrect END statement.";
   69:                         case SyntaxErrorType::expEndFor:       // 17
   70:                                 return "Expected END FOR, got {0}. Reached the end of a statement block,
 but there was an incorrect END statement.";
   71:                         case SyntaxErrorType::expEndWhile:      // 18
   72:                                 return "Expected END WHILE, got {0}. Reached the end of a statement bloc
k, but there was an incorrect END statement.";
   73:                         case SyntaxErrorType::expEndFunc:             // 19
   74:                                 return "Expected END FUNC, got {0}. Reached the end of a statement block
, but there was an incorrect END statement.";
   75:                         case SyntaxErrorType::expEndProc:             // 20
   76:                                 return "Expected END PROC, got {0}. Reached the end of a statement block
, but there was an incorrect END statement.";
```

```
   77:                         case SyntaxErrorType::expEndStruct:     // 21
   78:                             return "Expected END STRUCT, got {0}. Reached the end of a statement blo
ck, but there was an incorrect END statement.";
   79:                         case SyntaxErrorType::expTypeMod:             // 22
   80:                             return "Expected type or type modifier, got {0}. Input type must be spec
ified and must be a primitive";
   81:                         case SyntaxErrorType::tooManyElse:      // 23
   82:                             return "Got a second ELSE in the if tree. There can only be one ELSE, an
d it must be last. Got {0}";
   83:                         case SyntaxErrorType::unexpEndFunc:     // 24
   84:                         case SyntaxErrorType::unexpEndProc:    // 25
   85:                         case SyntaxErrorType::unexpEndIf:             // 26
   86:                         case SyntaxErrorType::unexpEndFor:      // 27
   87:                         case SyntaxErrorType::unexpEndWhile:  // 28
   88:                         case SyntaxErrorType::unexpEndStruct:  // 29
   89:                         case SyntaxErrorType::unexpElseIf:      // 30
   90:                             return "Unexpected end statement. This error is not implemented so pleas
e submit a bug report with the code causing it if you get it.";
   91:                         case SyntaxErrorType::inconsistentDict:
   92:                             return "Inconsistend dictionary: All items must be dictionary pairs, or
all must be single atoms. It is not permissible to mix them. Got {0}";
   93:                         default:
   94:                             return "Unrecognised or unimplemented error code.";
   95:                         }
   96:                 }
   97:
   98:         private:
   99:                 std::string m_Filename;
  100:                 std::string m_Output;
  101:                 std::shared_ptr<TranspilerState> m_State;
  102:                 bool m_PrintTokenList;
  103:         };
  104: }
```

```
    1: #pragma once
    2:
    3: #ifndef LOG_H_SUPERNOVA_CORE_GUARD
    4: #define LOG_H_SUPERNOVA_CORE_GUARD
    5:
    6: #include "spdlog/spdlog.h"
    7: #include "spdlog/fmt/ostr.h"
    8:
    9: namespace Spliwaca
   10: {
   11:         class Log
   12:         {
   13:         public:
   14:                 static void Init();
   15:                 inline static std::shared_ptr<spdlog::logger>& GetCoreLogger() { return s_CoreLogger; }
   16:                 inline static std::shared_ptr<spdlog::logger>& GetClientLogger() { return s_ClientLogger
; }
   17:
   18:         private:
   19:                 static std::shared_ptr<spdlog::logger> s_CoreLogger;
   20:                 static std::shared_ptr<spdlog::logger> s_ClientLogger;
   21:         };
   22: }
   23:
   24: #ifdef SPLW_DEBUG
   25:
   26: // Log macros
   27: #define SPLW_TRACE(...)         ::Spliwaca::Log::GetClientLogger()->trace(__VA_ARGS__)
   28: #define SPLW_INFO(...)          ::Spliwaca::Log::GetClientLogger()->info(__VA_ARGS__)
   29: #define SPLW_WARN(...)          ::Spliwaca::Log::GetClientLogger()->warn(__VA_ARGS__)
   30: #define SPLW_ERROR(...)         ::Spliwaca::Log::GetClientLogger()->error(__VA_ARGS__)
   31: #define SPLW_CRITICAL(...)      ::Spliwaca::Log::GetClientLogger()->critical(__VA_ARGS__)
   32: #define LOG_INIT()              ::Spliwaca::Log::Init()
   33:
   34: #else
   35: #ifdef SPLW_DIST
   36:
   37: #define SPLW_TRACE(...)         //::Spliwaca::Log::GetClientLogger()->trace(__VA_ARGS__)
   38: #define SPLW_INFO(...)          //::Spliwaca::Log::GetClientLogger()->info(__VA_ARGS__)
   39: #define SPLW_WARN(...)          ::Spliwaca::Log::GetClientLogger()->warn(__VA_ARGS__)
   40: #define SPLW_ERROR(...)         ::Spliwaca::Log::GetClientLogger()->error(__VA_ARGS__)
   41: #define SPLW_CRITICAL(...)      ::Spliwaca::Log::GetClientLogger()->critical(__VA_ARGS__)
   42: #define LOG_INIT()              ::Spliwaca::Log::Init()
   43:
   44: #else
   45: // Log macros
   46: #define SPLW_TRACE(...)         //::Spliwaca::Log::GetClientLogger()->trace(__VA_ARGS__)
   47: #define SPLW_INFO(...)
   48: #define SPLW_WARN(...)
   49: #define SPLW_ERROR(...)
   50: #define SPLW_CRITICAL(...)
   51: #define LOG_INIT()
   52:
   53: #endif
   54:
   55: #endif
   56:
   57: #endif /*LOG_H_SUPERNOVA_CORE_GUARD*/
```