```
    1: #include "SyntaxAnalyzer.h"
    2:
    3: SyntaxAnalyzer::SyntaxAnalyzer(const std::vector<Lexer::Token> &tokens, std::ofstr
eam &output, bool print) : tokens(tokens), it(tokens.begin()), currentToken(*(it)), outpu
t(output)
    4: {
    5:         this->print = print;
    6: }
    7:
    8: SyntaxAnalyzer::~SyntaxAnalyzer() { output.close(); }
    9:
   10: /**
   11:  * Get the next token in the list of tokens
   12:  * Increments iterator to current token
   13:  */
   14: void SyntaxAnalyzer::getNextToken()
   15: {
   16:     // Increment iterator
   17:         ++it;
   18:
   19:     if (it == this->tokens.end())
   20:     {
   21:         --it;
   22:         throw SyntaxError("Unexpected end of file", currentToken.lineNumber);
   23:     }
   24:
   25:     this->currentToken = *(it);
   26:
   27:     if (print)
   28:     {
   29:         printCurrentToken();
   30:     }
   31:
   32:         if (this->currentToken.token == "Illegal")
   33:         {
   34:                 throw SyntaxError("Illegal symbol \'" + this->currentToken.lexeme
+ "\'", this->currentToken.lineNumber);
   35:         }
   36: }
   37:
   38: // The root of the top-down parser
   39: void SyntaxAnalyzer::Rat18F()
   40: {
   41:     if (print)
   42:     {
   43:         printCurrentToken();
   44:         output << "\t<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration Li
st> <Statement List>" << std::endl;
   45:     }
   46:
   47:     OptFunctionDefinitions();
   48:
   49:     if (currentToken.lexeme == "$$")
   50:     {
   51:         getNextToken();
   52:         OptDeclarationList();
   53:         StatementList();
   54:     }
   55:
   56:     if (currentToken.lexeme != "$$")
   57:     {
   58:         throw SyntaxError("Expected '$$'.", currentToken.lineNumber);
   59:     }
   60: }
   61:
   62: void SyntaxAnalyzer::Parameter()
   63: {
   64:     if (print)
   65:     {
   66:         output << "\t<Parameter> -> <IDs> : <Qualifier>" << std::endl;
```

```
   67:     }
   68:
   69:     IDs();
   70:
   71:     if (currentToken.lexeme != ":")
   72:     {
   73:         throw SyntaxError("Expected ':'", currentToken.lineNumber);
   74:     }
   75:
   76:     getNextToken();
   77:     Qualifier();
   78: }
   79:
   80: void SyntaxAnalyzer::Function()
   81: {
   82:     if (print)
   83:     {
   84:         output << "\t<Function> ->  function <Identifier> ( <Opt Parameter List> )
 <Opt Declaration List> <Body>" << std::endl;
   85:     }
   86:
   87:     Identifier();
   88:
   89:     getNextToken();
   90:     if (currentToken.lexeme != "(")
   91:     {
   92:         throw SyntaxError("Expected '('", currentToken.lineNumber);
   93:     }
   94:
   95:     getNextToken();
   96:
   97:     OptParameterList();
   98:
   99:     if (currentToken.lexeme != ")")
  100:     {
  101:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
  102:     }
  103:
  104:     getNextToken();
  105:     OptDeclarationList();
  106:     Body();
  107: }
  108:
  109: void SyntaxAnalyzer::OptFunctionDefinitions()
  110: {
  111:     if (print)
  112:     {
  113:         output << "\t<Opt Function Definitions> ->  <Function Definitions> | <Empt
y>" << std::endl;
  114:     }
  115:
  116:     if (currentToken.lexeme == "function")
  117:     {
  118:         getNextToken();
  119:         FunctionDefinitions();
  120:     }
  121:     else
  122:     {
  123:         Empty();
  124:     }
  125: }
  126:
  127: void SyntaxAnalyzer::OptDeclarationList()
  128: {
  129:     if (print)
  130:     {
  131:         output << "\t<Opt Declaration List> -> <Declaration List> | <Empty>" << st
d::endl;
  132:     }
  133:
```

```
  134:     if (currentToken.lexeme == "real" | currentToken.lexeme == "boolean" | current
Token.lexeme == "int")
  135:     {
  136:         DeclarationList();
  137:     }
  138:     else
  139:     {
  140:         Empty();
  141:     }
  142: }
  143:
  144: void SyntaxAnalyzer::DeclarationList()
  145: {
  146:     if (print)
  147:     {
  148:         output << "\t<Declaration List> -> <Declaration>; | <Declaration>; <Declar
ation List>\n";
  149:     }
  150:
  151:     Declaration();
  152:
  153:     if (currentToken.lexeme == ";")
  154:     {
  155:         getNextToken();
  156:         if (currentToken.lexeme == "real" | currentToken.lexeme == "boolean" | cur
rentToken.lexeme == "int")
  157:         {
  158:             DeclarationList();
  159:         }
  160:     }
  161: }
  162:
  163: void SyntaxAnalyzer::Declaration()
  164: {
  165:     if (print)
  166:     {
  167:         output << "\t<Declaration> -> <Qualifier> <IDs>" << std::endl;
  168:     }
  169:
  170:     Qualifier();
  171:     getNextToken();
  172:
  173:     if (currentToken.token == "Identifier")
  174:     {
  175:         IDs();
  176:     }
  177: }
  178:
  179: void SyntaxAnalyzer::Qualifier()
  180: {
  181:     if (print)
  182:     {
  183:         output << "\t<Qualifier> ->  int | boolean | real" << std::endl;
  184:     }
  185: }
  186:
  187: void SyntaxAnalyzer::IDs()
  188: {
  189:     if (print)
  190:     {
  191:         output << "\t<IDs> -> <Identifier> | <Identifier>, <IDs>" << std::endl;
  192:     }
  193:
  194:     Identifier();
  195:     getNextToken();
  196:
  197:     if (currentToken.lexeme == ",")
  198:     {
  199:         getNextToken();
  200:         if (currentToken.token == "Identifier")
```

```
201:         {
202:             IDs();
203:         }
204:         else
205:         {
206:             throw SyntaxError("Expected identifier", currentToken.lineNumber);
207:         }
208:     }
209: }
210:
211: void SyntaxAnalyzer::Identifier()
212: {
213:     if (print)
214:     {
215:         output << "\t<Identifier>" << std::endl;
216:     }
217: }
218:
219: void SyntaxAnalyzer::StatementList()
220: {
221:     if (print)
222:     {
223:         output << "\t<Statement List> -> <Statement> | <Statement> <Statement List
>" << std::endl;
224:     }
225:
226:     Statement();
227:
228:     if (currentToken.lexeme == "get" | currentToken.lexeme == "put" | currentToken
.lexeme == "while" | currentToken.lexeme == "if" |
229:         currentToken.lexeme == "return" | currentToken.token == "Identifier")
230:     {
231:         StatementList();
232:     }
233: }
234:
235: void SyntaxAnalyzer::Statement()
236: {
237:     if (print)
238:     {
239:         output << "\t<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Pri
nt> | <Scan> | <While>" << std::endl;
240:     }
241:
242:     if (currentToken.lexeme == "{")
243:     {
244:         getNextToken();
245:         Compound();
246:     }
247:     else if (currentToken.token == "Identifier")
248:     {
249:         Assign();
250:     }
251:     else if (currentToken.lexeme == "if")
252:     {
253:         getNextToken();
254:         If();
255:     }
256:     else if (currentToken.lexeme == "return")
257:     {
258:         getNextToken();
259:         Return();
260:     }
261:     else if (currentToken.lexeme == "put")
262:     {
263:         getNextToken();
264:         Print();
265:     }
266:     else if (currentToken.lexeme == "get")
267:     {
```

```
  268:         getNextToken();
  269:         Scan();
  270:     }
  271:     else if (currentToken.lexeme == "while")
  272:     {
  273:         getNextToken();
  274:         While();
  275:     }
  276:     else
  277:     {
  278:         throw SyntaxError("Expected '{', identifier or keyword", currentToken.line
Number);
  279:     }
  280: }
  281:
  282: void SyntaxAnalyzer::Compound()
  283: {
  284:     if (print)
  285:     {
  286:         output << "\t<Compound> -> { <Statement List> }" << std::endl;
  287:     }
  288:
  289:     StatementList();
  290:
  291:     if (currentToken.lexeme != "}")
  292:     {
  293:         throw SyntaxError("Expected '}'", currentToken.lineNumber);
  294:     }
  295:
  296:     getNextToken();
  297: }
  298:
  299: void SyntaxAnalyzer::Assign()
  300: {
  301:     if (print)
  302:     {
  303:         output << "\t<Assign> -> <Identifier> = <Expression>;" << std::endl;
  304:     }
  305:
  306:     Identifier();
  307:
  308:     getNextToken();
  309:
  310:     if (currentToken.lexeme != "=")
  311:     {
  312:         throw SyntaxError("Expected '='", currentToken.lineNumber);
  313:     }
  314:
  315:     getNextToken();
  316:     Expression();
  317:
  318:     if (currentToken.lexeme != ";")
  319:     {
  320:         throw SyntaxError("Expected ';'", currentToken.lineNumber);
  321:     }
  322:     getNextToken();
  323: }
  324:
  325: void SyntaxAnalyzer::Expression()
  326: {
  327:     if (print)
  328:     {
  329:         output << "\t<Expression> -> <Term> <ExpressionPrime>" << std::endl;
  330:     }
  331:
  332:     Term();
  333:     ExpressionPrime();
  334: }
  335:
  336: void SyntaxAnalyzer::ExpressionPrime()
```

```
337: {
338:     if (print)
339:     {
340:         output << "\t<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <E
xpressionPrime> | <Empty>" << std::endl;
341:     }
342:
343:     if (currentToken.lexeme == "+" | currentToken.lexeme == "-")
344:     {
345:         getNextToken();
346:
347:         Term();
348:         ExpressionPrime();
349:     }
350:     else
351:     {
352:         Empty();
353:     }
354: }
355:
356: void SyntaxAnalyzer::Term()
357: {
358:     if (print)
359:     {
360:         output << "\t<Term> -> <Factor> <TermPrime>" << std::endl;
361:     }
362:
363:     Factor();
364:     TermPrime();
365: }
366:
367: void SyntaxAnalyzer::Factor()
368: {
369:     if (print)
370:     {
371:         output << "\t<Factor> -> - <Primary> | <Primary>" << std::endl;
372:     }
373:
374:     if (currentToken.lexeme == "-")
375:     {
376:         getNextToken();
377:     }
378:
379:     Primary();
380: }
381:
382: void SyntaxAnalyzer::Primary()
383: {
384:     if (print)
385:     {
386:         output << "\t<Primary> -> <Identifier> | <Integer> | <Identifier> ( <IDs>
) | ( <Expression> ) | <Real> | true | false" << std::endl;
387:     }
388:
389:     if (currentToken.token == "Identifier")
390:     {
391:         Identifier();
392:
393:         getNextToken();
394:         if (currentToken.lexeme == "(")
395:         {
396:             getNextToken();
397:             IDs();
398:
399:             if (currentToken.lexeme != ")")
400:             {
401:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
402:             }
403:
404:             getNextToken();
```

```
405:         }
406:     }
407:     else if (currentToken.token == "Integer")
408:     {
409:         Integer();
410:         getNextToken();
411:     }
412:     else if (currentToken.lexeme == "(")
413:     {
414:         getNextToken();
415:
416:         Expression();
417:
418:         if (currentToken.lexeme != ")")
419:         {
420:             throw SyntaxError("Expected ')'", currentToken.lineNumber);
421:         }
422:         getNextToken();
423:     }
424:     else if (currentToken.token == "Real")
425:     {
426:         Real();
427:         getNextToken();
428:     }
429:     else if (currentToken.lexeme == "true")
430:     {
431:         output << "\ttrue" << std::endl;
432:         getNextToken();
433:     }
434:     else if (currentToken.lexeme == "false")
435:     {
436:         output << "\tfalse" << std::endl;
437:         getNextToken();
438:     }
439: }
440:
441: void SyntaxAnalyzer::Integer()
442: {
443:     if (print)
444:     {
445:         output << "\t<Integer>" << std::endl;
446:     }
447: }
448:
449: void SyntaxAnalyzer::Real()
450: {
451:     if (print)
452:     {
453:         output << "\t<Real>" << std::endl;
454:     }
455: }
456:
457: void SyntaxAnalyzer::Return()
458: {
459:     if (print)
460:     {
461:         output << "\t<Return> -> return; |  return <Expression>;" << std::endl;
462:     }
463:
464:     if (currentToken.lexeme != ";")
465:     {
466:         Expression();
467:     }
468:     getNextToken();
469: }
470:
471: void SyntaxAnalyzer::If()
472: {
473:     if (print)
474:     {
```

```
  475:         output << "\t<If> -> if ( <Condition> ) <Statement> endif | if ( <Conditio
n> ) <Statement> else <Statement> endif" << std::endl;
  476:     }
  477:
  478:     if (currentToken.lexeme != "(")
  479:     {
  480:         throw SyntaxError("Expected '('", currentToken.lineNumber);
  481:     }
  482:
  483:         getNextToken();
  484:
  485:     Condition();
  486:
  487:     if (currentToken.lexeme != ")")
  488:     {
  489:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
  490:     }
  491:
  492:     Statement();
  493:
  494:     getNextToken();
  495:
  496:     if (currentToken.lexeme == "else")
  497:     {
  498:         Statement();
  499:     }
  500:
  501:     getNextToken();
  502:
  503:     if (currentToken.lexeme != "ifend")
  504:     {
  505:         throw SyntaxError("Expected 'ifend' keyword", currentToken.lineNumber);
  506:     }
  507: }
  508:
  509: void SyntaxAnalyzer::Condition()
  510: {
  511:     if (print)
  512:     {
  513:         output << "\t<Condition> -> <Expression>  <Relop>  <Expression>" << std::e
ndl;
  514:     }
  515:
  516:     Expression();
  517:
  518:     Relop();
  519:
  520:     getNextToken();
  521:     Expression();
  522: }
  523:
  524: void SyntaxAnalyzer::Relop()
  525: {
  526:     if (currentToken.lexeme != "==" && currentToken.lexeme != "^=" && currentToken
.lexeme != ">" && currentToken.lexeme != "<" && currentToken.lexeme != "=>" && currentTok
en.lexeme != "=<")
  527:     {
  528:         throw SyntaxError("Expected relational operator", currentToken.lineNumber)
;
  529:     }
  530:
  531:     if (print)
  532:     {
  533:         output << "\t<Relop> -> " << currentToken.lexeme << std::endl;
  534:     }
  535: }
  536:
  537: void SyntaxAnalyzer::Empty()
  538: {
  539:     if (print)
```

```
540:     {
541:         output << "\t<Empty> -> Îµ" << std::endl;
542:     }
543: }
544:
545: void SyntaxAnalyzer::Body()
546: {
547:     if (print)
548:     {
549:         output << "\t<Body> -> { <Statement List> }" << std::endl;
550:     }
551:
552:     if (currentToken.lexeme != "{")
553:     {
554:         throw SyntaxError("Expected '{'", currentToken.lineNumber);
555:     }
556:
557:     getNextToken();
558:
559:     StatementList();
560:
561:     if (currentToken.lexeme != "}")
562:     {
563:         throw SyntaxError("Expected '}'", currentToken.lineNumber);
564:     }
565:
566:     getNextToken();
567: }
568:
569: void SyntaxAnalyzer::FunctionDefinitions()
570: {
571:     if (print)
572:     {
573:         output << "\t<Function Definitions> -> <Function> | <Function> <Function D
efinitions>" << std::endl;
574:     }
575:
576:     Function();
577:
578:     if (currentToken.lexeme == "function")
579:     {
580:         getNextToken();
581:         FunctionDefinitions();
582:     }
583: }
584:
585: void SyntaxAnalyzer::Print()
586: {
587:     if (print)
588:     {
589:         output << "\t<Print> -> put ( <Expression> );" << std::endl;
590:     }
591:
592:     if (currentToken.lexeme != "(")
593:     {
594:         throw SyntaxError("Expected '('", currentToken.lineNumber);
595:     }
596:
597:     getNextToken();
598:     Expression();
599:
600:     if (currentToken.lexeme != ")")
601:     {
602:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
603:     }
604:     getNextToken();
605:
606:     if (currentToken.lexeme != ";")
607:     {
608:         throw SyntaxError("Expected ';'", currentToken.lineNumber);
```

```
609:     }
610:
611:     getNextToken();
612: }
613:
614: void SyntaxAnalyzer::Scan()
615: {
616:     if (print)
617:     {
618:         output << "\t<Scan> -> get ( <IDs> );" << std::endl;
619:     }
620:
621:     if (currentToken.lexeme != "(")
622:     {
623:         throw SyntaxError("Expected '('", currentToken.lineNumber);
624:     }
625:
626:     getNextToken();
627:     IDs();
628:
629:     if (currentToken.lexeme != ")")
630:     {
631:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
632:     }
633:
634:     getNextToken();
635:     if (currentToken.lexeme != ";")
636:     {
637:         throw SyntaxError("Expected ';'", currentToken.lineNumber);
638:     }
639:
640:     getNextToken();
641: }
642:
643: void SyntaxAnalyzer::TermPrime()
644: {
645:     if (print)
646:     {
647:         output << "\t<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime
> | <Empty>" << std::endl;
648:     }
649:
650:     if (currentToken.lexeme == "*" | currentToken.lexeme == "/")
651:     {
652:         getNextToken();
653:
654:         Factor();
655:         TermPrime();
656:     }
657: }
658:
659: /**
660:  * Attempt to syntactically analyze a list of
661:  * Lexer tokens
662:  */
663: void SyntaxAnalyzer::Analyze()
664: {
665:     Rat18F();
666:     output << "Syntax Analysis Successful." << std::endl;
667: }
668:
669: void SyntaxAnalyzer::OptParameterList()
670: {
671:     if (print)
672:     {
673:         output << "\t<Opt Parameter List> -> <Parameter List> | <Empty>" << std::e
ndl;
674:     }
675:
676:     if (currentToken.lexeme == ")")
```

```
677:     {
678:         Empty();
679:     }
680:     else if (currentToken.token == "Identifier")
681:     {
682:         ParameterList();
683:     }
684:     else
685:     {
686:         throw SyntaxError("Expected ')' or identifier", currentToken.lineNumber);
687:     }
688: }
689:
690: void SyntaxAnalyzer::ParameterList()
691: {
692:     if (print)
693:     {
694:         output << "\t<Parameter List> -> <Parameter> | <Parameter> , <Parameter Li
st>" << std::endl;
695:     }
696:
697:     Parameter();
698:
699:     getNextToken();
700:
701:     if (currentToken.lexeme == ",")
702:     {
703:         getNextToken();
704:         ParameterList();
705:     }
706: }
707:
708: void SyntaxAnalyzer::While()
709: {
710:     if (print)
711:     {
712:         output << "\t<While> -> while ( <Condition> )  <Statement>" << std::endl;
713:     }
714:
715:     if (currentToken.lexeme != "(")
716:     {
717:         throw SyntaxError("Expected '('", currentToken.lineNumber);
718:     }
719:     getNextToken();
720:
721:     Condition();
722:
723:     if (currentToken.lexeme != ")")
724:     {
725:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
726:     }
727:     getNextToken();
728:     Statement();
729:
730:     if (currentToken.lexeme != "whileend")
731:     {
732:         throw SyntaxError("Expected 'whileend' keyword", currentToken.lineNumber);
733:     }
734:     getNextToken();
735: }
736:
737: void SyntaxAnalyzer::printCurrentToken()
738: {
739:     output << std::left << std::endl
740:                 << std::setw(8) << "Token:" << std::setw(16) << currentToken.token <
< std::setw(8) << "Lexeme:" << currentToken.lexeme << std::endl
741:                 << std::endl;
742: }
743:
744: SyntaxError::SyntaxError(std::string message, int lineNumber)
```

```
745: {
746:     this->message = message;
747:     this->lineNumber = lineNumber;
748: }
749:
750: SyntaxError::~SyntaxError() {}
751:
752: std::string SyntaxError::getMessage() const
753: {
754:     return (this->message + " Line: " + std::to_string(this->lineNumber));
755: }
```