```
    1: #include "SyntaxAnalyzer.h"
    2:
    3: SyntaxAnalyzer::SyntaxAnalyzer(const std::vector<Lexer::Token> &tokens, std::ofstream &output, bool print) : tokens(toke
ns), it(tokens.begin()), currentToken(*(it)), output(output), save(nullptr)
    4: {
    5:         this->print = print;
    6:         this->save = new Lexer::Token();
    7:         this->errCount = 0;
    8:         this->isDeclaration = false;
    9: }
   10:
   11: SyntaxAnalyzer::~SyntaxAnalyzer()
   12: {
   13: }
   14:
   15: void SyntaxAnalyzer::error(ErrorType errorType, int lineNumber, std::string expected)
   16: {
   17:         errCount++;
   18:         err << "[ERR] (Line " << lineNumber << ") ";
   19:         switch (errorType)
   20:         {
   21:         case TYPE_MISMATCH:
   22:         {
   23:                 err << "TYPE MISMATCH";
   24:                 if (expected != "")
   25:                 {
   26:                         err << ". Expected \"" << expected << "\"";
   27:                 }
   28:                 break;
   29:         }
   30:         case DUPLICATE_SYMBOL:
   31:         {
   32:                 err << "DUPLICATE SYMBOL";
   33:                 if (expected != "")
   34:                 {
   35:                         err << " \"" << expected << "\"";
   36:                 }
   37:                 break;
   38:         }
   39:         case UNDECLARED_VARIABLE:
   40:         {
   41:                 err << "UNDECLARED VARIABLE";
   42:                 if (expected != "")
   43:                 {
   44:                         err << " \"" << expected << "\"";
   45:                 }
   46:
   47:                 break;
   48:         }
   49:         }
   50:         err << std::endl;
   51: }
   52:
   53: /**
   54:  * Get the next token in the list of tokens
   55:  * Increments iterator to current token
   56:  */
   57: void SyntaxAnalyzer::getNextToken()
   58: {
   59:         // Increment iterator
   60:         ++it;
   61:
   62:         if (it == this->tokens.end())
   63:         {
   64:                 --it;
   65:                 throw SyntaxError("Unexpected end of file", currentToken.lineNumber);
   66:         }
   67:
   68:         this->currentToken = *(it);
   69:
   70:         if (print)
   71:         {
   72:                 printCurrentToken();
   73:         }
   74:
   75:         if (this->currentToken.token == "Illegal")
   76:         {
   77:                 throw SyntaxError("Illegal symbol \'" + this->currentToken.lexeme + "\'", this->currentToken.lineNumber)
;
   78:         }
   79: }
   80:
   81: // The root of the top-down parser
   82: void SyntaxAnalyzer::Rat18F()
   83: {
   84:         if (print)
   85:         {
   86:                 printCurrentToken();
   87:                 output << "\t<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List>" << std::
endl;
   88:         }
   89:
   90:         OptFunctionDefinitions();
   91:
   92:         if (currentToken.lexeme == "$$")
   93:         {
```

```
   94:                 getNextToken();
   95:                 OptDeclarationList();
   96:                 StatementList();
   97:         }
   98:
   99:         if (currentToken.lexeme != "$$")
  100:         {
  101:                 throw SyntaxError("Expected '$$'.", currentToken.lineNumber);
  102:         }
  103: }
  104:
  105: void SyntaxAnalyzer::Parameter()
  106: {
  107:         if (print)
  108:         {
  109:                 output << "\t<Parameter> -> <IDs> : <Qualifier>" << std::endl;
  110:         }
  111:
  112:         IDs();
  113:
  114:         if (currentToken.lexeme != ":")
  115:         {
  116:                 throw SyntaxError("Expected ':'", currentToken.lineNumber);
  117:         }
  118:
  119:         getNextToken();
  120:         Qualifier();
  121: }
  122:
  123: void SyntaxAnalyzer::Function()
  124: {
  125:         if (print)
  126:         {
  127:                 output << "\t<Function> ->  function <Identifier> ( <Opt Parameter List> ) <Opt Declaration List> <Body>
" << std::endl;
  128:         }
  129:
  130:         Identifier();
  131:
  132:         getNextToken();
  133:         if (currentToken.lexeme != "(")
  134:         {
  135:                 throw SyntaxError("Expected '('", currentToken.lineNumber);
  136:         }
  137:
  138:         getNextToken();
  139:
  140:         OptParameterList();
  141:
  142:         if (currentToken.lexeme != ")")
  143:         {
  144:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
  145:         }
  146:
  147:         getNextToken();
  148:         OptDeclarationList();
  149:         Body();
  150: }
  151:
  152: void SyntaxAnalyzer::OptFunctionDefinitions()
  153: {
  154:         if (print)
  155:         {
  156:                 output << "\t<Opt Function Definitions> ->  <Function Definitions> | <Empty>" << std::endl;
  157:         }
  158:
  159:         if (currentToken.lexeme == "function")
  160:         {
  161:                 getNextToken();
  162:                 FunctionDefinitions();
  163:         }
  164:         else
  165:         {
  166:                 Empty();
  167:         }
  168: }
  169:
  170: void SyntaxAnalyzer::OptDeclarationList()
  171: {
  172:         if (print)
  173:         {
  174:                 output << "\t<Opt Declaration List> -> <Declaration List> | <Empty>" << std::endl;
  175:         }
  176:
  177:         if (currentToken.lexeme == "real" | currentToken.lexeme == "boolean" | currentToken.lexeme == "int")
  178:         {
  179:                 DeclarationList();
  180:         }
  181:         else
  182:         {
  183:                 Empty();
  184:         }
  185: }
  186:
  187: void SyntaxAnalyzer::DeclarationList()
  188: {
```

```
189:         if (print)
190:         {
191:                 output << "\t<Declaration List> -> <Declaration>; | <Declaration>; <Declaration List>\n";
192:         }
193:
194:         // Save variable type
195:         savedType = new std::string(currentToken.lexeme);
196:
197:         this->isDeclaration = true;
198:
199:         Declaration();
200:
201:         if (currentToken.lexeme == ";")
202:         {
203:                 // Done with declaration, pop type stack
204:                 symbolTable.pop_typestack();
205:
206:                 getNextToken();
207:                 if (currentToken.lexeme == "real" | currentToken.lexeme == "boolean" | currentToken.lexeme == "int")
208:                 {
209:                         DeclarationList();
210:                 }
211:         }
212:
213:         this->isDeclaration = false;
214: }
215:
216: void SyntaxAnalyzer::Declaration()
217: {
218:         if (print)
219:         {
220:                 output << "\t<Declaration> -> <Qualifier> <IDs>" << std::endl;
221:         }
222:
223:         Qualifier();
224:         getNextToken();
225:
226:         if (currentToken.token == "Identifier")
227:         {
228:                 IDs();
229:         }
230: }
231:
232: void SyntaxAnalyzer::Qualifier()
233: {
234:         if (print)
235:         {
236:                 output << "\t<Qualifier> ->  int | boolean | real" << std::endl;
237:         }
238: }
239:
240: void SyntaxAnalyzer::IDs()
241: {
242:         if (print)
243:         {
244:                 output << "\t<IDs> -> <Identifier> | <Identifier>, <IDs>" << std::endl;
245:         }
246:
247:         if (isDeclaration) {
248:                 if (!symbolTable.lookup(currentToken))
249:                 {
250:                         symbolTable.insert(currentToken, *savedType);
251:                 }
252:                 else
253:                 {
254:                         error(DUPLICATE_SYMBOL, currentToken.lineNumber, currentToken.lexeme);
255:                 }
256:         }
257:
258:         Identifier();
259:         getNextToken();
260:
261:         if (currentToken.lexeme == ",")
262:         {
263:                 getNextToken();
264:                 if (currentToken.token == "Identifier")
265:                 {
266:                         IDs();
267:                 }
268:                 else
269:                 {
270:                         throw SyntaxError("Expected identifier", currentToken.lineNumber);
271:                 }
272:         }
273: }
274:
275: void SyntaxAnalyzer::Identifier()
276: {
277:         if (print)
278:         {
279:                 output << "\t<Identifier>" << std::endl;
280:         }
281: }
282:
283: void SyntaxAnalyzer::StatementList()
284: {
```

```
  285:         if (print)
  286:         {
  287:                 output << "\t<Statement List> -> <Statement> | <Statement> <Statement List>" << std::endl;
  288:         }
  289:
  290:         Statement();
  291:
  292:         if (currentToken.lexeme == "get" | currentToken.lexeme == "put" | currentToken.lexeme == "while" | currentToken.
lexeme == "if" |
  293:                 currentToken.lexeme == "return" | currentToken.token == "Identifier")
  294:         {
  295:                 StatementList();
  296:         }
  297: }
  298:
  299: void SyntaxAnalyzer::Statement()
  300: {
  301:         if (print)
  302:         {
  303:                 output << "\t<Statement> -> <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>" << std
::endl;
  304:         }
  305:
  306:         if (currentToken.lexeme == "{")
  307:         {
  308:                 getNextToken();
  309:                 Compound();
  310:         }
  311:         else if (currentToken.token == "Identifier")
  312:         {
  313:                 Assign();
  314:         }
  315:         else if (currentToken.lexeme == "if")
  316:         {
  317:                 getNextToken();
  318:                 If();
  319:         }
  320:         else if (currentToken.lexeme == "return")
  321:         {
  322:                 getNextToken();
  323:                 Return();
  324:         }
  325:         else if (currentToken.lexeme == "put")
  326:         {
  327:                 getNextToken();
  328:                 Print();
  329:         }
  330:         else if (currentToken.lexeme == "get")
  331:         {
  332:                 getNextToken();
  333:                 Scan();
  334:         }
  335:         else if (currentToken.lexeme == "while")
  336:         {
  337:                 getNextToken();
  338:                 While();
  339:         }
  340:         else
  341:         {
  342:                 throw SyntaxError("Expected '{', identifier or keyword", currentToken.lineNumber);
  343:         }
  344: }
  345:
  346: void SyntaxAnalyzer::Compound()
  347: {
  348:         if (print)
  349:         {
  350:                 output << "\t<Compound> -> { <Statement List> }" << std::endl;
  351:         }
  352:
  353:         StatementList();
  354:
  355:         if (currentToken.lexeme != "}")
  356:         {
  357:                 throw SyntaxError("Expected '}'", currentToken.lineNumber);
  358:         }
  359:
  360:         getNextToken();
  361: }
  362:
  363: void SyntaxAnalyzer::Assign()
  364: {
  365:         if (print)
  366:         {
  367:                 output << "\t<Assign> -> <Identifier> = <Expression>;" << std::endl;
  368:         }
  369:
  370:         Identifier();
  371:
  372:         // Save the value of the current token to gen instruction later
  373:         *save = currentToken;
  374:         std::string type = symbolTable.get_type(*save);
  375:         if (type == "")
  376:         {
  377:                 error(UNDECLARED_VARIABLE, currentToken.lineNumber, currentToken.lexeme);
  378:         }
```

```
  379:         else
  380:         {
  381:                 symbolTable.push_typestack(symbolTable.get_type(*save));
  382:         }
  383:
  384:         getNextToken();
  385:
  386:         if (currentToken.lexeme != "=")
  387:         {
  388:                 throw SyntaxError("Expected '='", currentToken.lineNumber);
  389:         }
  390:
  391:         getNextToken();
  392:         Expression();
  393:
  394:         symbolTable.gen_instr("POPM", symbolTable.get_address(*save));
  395:
  396:         if (currentToken.lexeme != ";")
  397:         {
  398:                 throw SyntaxError("Expected ';'", currentToken.lineNumber);
  399:         }
  400:
  401:         symbolTable.pop_typestack();
  402:
  403:         getNextToken();
  404: }
  405:
  406: void SyntaxAnalyzer::Expression()
  407: {
  408:         if (print)
  409:         {
  410:                 output << "\t<Expression> -> <Term> <ExpressionPrime>" << std::endl;
  411:         }
  412:
  413:         Term();
  414:         ExpressionPrime();
  415: }
  416:
  417: void SyntaxAnalyzer::ExpressionPrime()
  418: {
  419:         if (print)
  420:         {
  421:                 output << "\t<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <Empty>" <<
std::endl;
  422:         }
  423:
  424:         if (currentToken.lexeme == "+" | currentToken.lexeme == "-")
  425:         {
  426:                 std::string op = currentToken.lexeme;
  427:                 getNextToken();
  428:
  429:                 Term();
  430:
  431:                 if (op == "+")
  432:                 {
  433:                         symbolTable.gen_instr("ADD", NIL);
  434:                 }
  435:                 else
  436:                 {
  437:                         symbolTable.gen_instr("SUB", NIL);
  438:                 }
  439:                 ExpressionPrime();
  440:         }
  441:         else
  442:         {
  443:                 Empty();
  444:         }
  445: }
  446:
  447: void SyntaxAnalyzer::Term()
  448: {
  449:         if (print)
  450:         {
  451:                 output << "\t<Term> -> <Factor> <TermPrime>" << std::endl;
  452:         }
  453:
  454:         Factor();
  455:         TermPrime();
  456: }
  457:
  458: void SyntaxAnalyzer::Factor()
  459: {
  460:         if (print)
  461:         {
  462:                 output << "\t<Factor> -> - <Primary> | <Primary>" << std::endl;
  463:         }
  464:
  465:         if (currentToken.lexeme == "-")
  466:         {
  467:                 getNextToken();
  468:         }
  469:
  470:         Primary();
  471: }
  472:
  473: void SyntaxAnalyzer::Primary()
```

```
  474: {
  475:         if (print)
  476:         {
  477:                 output << "\t<Primary> -> <Identifier> | <Integer> | <Identifier> ( <IDs> ) | ( <Expression> ) | <Real>
| true | false" << std::endl;
  478:         }
  479:
  480:         if (currentToken.token == "Identifier")
  481:         {
  482:                 // If typestack is empty, we should be within a Condition.
  483:                 // Push the current Identifier's type onto the stack to compare with
  484:                 // the next one we see.
  485:                 if (symbolTable.typestack_empty())
  486:                 {
  487:                         symbolTable.push_typestack(*savedType);
  488:                 }
  489:                 // If the Identifier doesn't have a type, it isn't in the symbol table.
  490:                 else if (symbolTable.get_type(currentToken) == "")
  491:                 {
  492:                         error(UNDECLARED_VARIABLE, currentToken.lineNumber, currentToken.lexeme);
  493:                 }
  494:                 // Error TYPE MISMATCH
  495:                 else if (symbolTable.get_type(currentToken) != symbolTable.top_typestack())
  496:                 {
  497:                         error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
  498:                 }
  499:
  500:                 Identifier();
  501:                 symbolTable.gen_instr("PUSHM", symbolTable.lookup(currentToken));
  502:
  503:                 getNextToken();
  504:                 if (currentToken.lexeme == "(")
  505:                 {
  506:                         getNextToken();
  507:                         IDs();
  508:
  509:                         if (currentToken.lexeme != ")")
  510:                         {
  511:                                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
  512:                         }
  513:
  514:                         getNextToken();
  515:                 }
  516:         }
  517:         else if (currentToken.token == "Integer")
  518:         {
  519:                 // ERROR: TYPE MISMATCH
  520:                 if (symbolTable.top_typestack() != "int")
  521:                 {
  522:                         error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
  523:                 }
  524:
  525:                 Integer();
  526:                 symbolTable.gen_instr("PUSHI", stoi(currentToken.lexeme));
  527:                 getNextToken();
  528:         }
  529:         else if (currentToken.lexeme == "(")
  530:         {
  531:                 getNextToken();
  532:
  533:                 Expression();
  534:
  535:                 if (currentToken.lexeme != ")")
  536:                 {
  537:                         throw SyntaxError("Expected ')'", currentToken.lineNumber);
  538:                 }
  539:                 getNextToken();
  540:         }
  541:         else if (currentToken.token == "Real")
  542:         {
  543:                 // Error TYPE MISMATCH
  544:                 if (symbolTable.top_typestack() != "real")
  545:                 {
  546:                         error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
  547:                 }
  548:
  549:                 Real();
  550:                 getNextToken();
  551:         }
  552:         else if (currentToken.lexeme == "true")
  553:         {
  554:                 // Error TYPE MISMATCH
  555:                 if (symbolTable.top_typestack() != "boolean")
  556:                 {
  557:                         error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
  558:                 }
  559:
  560:                 if (print)
  561:                 {
  562:                         output << "\ttrue" << std::endl;
  563:                 }
  564:
  565:                 symbolTable.gen_instr("PUSHI", 1);
  566:                 getNextToken();
  567:         }
  568:         else if (currentToken.lexeme == "false")
```

```
  569:         {
  570:                 // Error TYPE MISMATCH
  571:                 if (symbolTable.top_typestack() != "boolean")
  572:                 {
  573:                         error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
  574:                 }
  575:
  576:                 if (print)
  577:                 {
  578:                         output << "\tfalse" << std::endl;
  579:                 }
  580:                 symbolTable.gen_instr("PUSHI", 0);
  581:                 getNextToken();
  582:         }
  583: }
  584:
  585: void SyntaxAnalyzer::Integer()
  586: {
  587:         if (print)
  588:         {
  589:                 output << "\t<Integer>" << std::endl;
  590:         }
  591: }
  592:
  593: void SyntaxAnalyzer::Real()
  594: {
  595:         if (print)
  596:         {
  597:                 output << "\t<Real>" << std::endl;
  598:         }
  599: }
  600:
  601: void SyntaxAnalyzer::Return()
  602: {
  603:         if (print)
  604:         {
  605:                 output << "\t<Return> -> return; |  return <Expression>;" << std::endl;
  606:         }
  607:
  608:         if (currentToken.lexeme != ";")
  609:         {
  610:                 Expression();
  611:         }
  612:         getNextToken();
  613: }
  614:
  615: void SyntaxAnalyzer::If()
  616: {
  617:         if (print)
  618:         {
  619:                 output << "\t<If> -> if ( <Condition> ) <Statement> endif | if ( <Condition> ) <Statement> else <Stateme
nt> endif" << std::endl;
  620:         }
  621:
  622:         if (currentToken.lexeme != "(")
  623:         {
  624:                 throw SyntaxError("Expected '('", currentToken.lineNumber);
  625:         }
  626:
  627:         getNextToken();
  628:
  629:         Condition();
  630:
  631:         if (currentToken.lexeme != ")")
  632:         {
  633:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
  634:         }
  635:
  636:         getNextToken();
  637:
  638:         Statement();
  639:
  640:         if (currentToken.lexeme == "else")
  641:         {
  642:                 Statement();
  643:         }
  644:
  645:         if (currentToken.lexeme != "ifend")
  646:         {
  647:                 throw SyntaxError("Expected 'ifend' keyword", currentToken.lineNumber);
  648:         }
  649:
  650:         symbolTable.back_patch(symbolTable.get_instr_address());
  651:         getNextToken();
  652: }
  653:
  654: void SyntaxAnalyzer::Condition()
  655: {
  656:         if (print)
  657:         {
  658:                 output << "\t<Condition> -> <Expression>  <Relop>  <Expression>" << std::endl;
  659:         }
  660:
  661:         // Save variable type to push to typestack later
  662:         savedType = new std::string(symbolTable.get_type(currentToken));
  663:
```

```
  664:         if (*savedType == "")
  665:         {
  666:                 error(UNDECLARED_VARIABLE, currentToken.lineNumber, currentToken.lexeme);
  667:         }
  668:
  669:         Expression();
  670:
  671:         Relop();
  672:
  673:         getNextToken();
  674:         Expression();
  675:
  676:         if (*savedOp == "<")
  677:         {
  678:                 symbolTable.gen_instr("LES", NIL);
  679:         }
  680:         else if (*savedOp == ">")
  681:         {
  682:                 symbolTable.gen_instr("GRT", NIL);
  683:         }
  684:         else if (*savedOp == "==")
  685:         {
  686:                 symbolTable.gen_instr("EQU", NIL);
  687:         }
  688:         else if (*savedOp == "^=")
  689:         {
  690:                 symbolTable.gen_instr("NEQ", NIL);
  691:         }
  692:         else if (*savedOp == "=>")
  693:         {
  694:                 symbolTable.gen_instr("GEQ", NIL);
  695:         }
  696:         else if (*savedOp == "=<")
  697:         {
  698:                 symbolTable.gen_instr("LEQ", NIL);
  699:         }
  700:
  701:         symbolTable.push_jumpstack(symbolTable.get_instr_address());
  702:         symbolTable.gen_instr("JUMPZ", NIL);
  703: }
  704:
  705: void SyntaxAnalyzer::Relop()
  706: {
  707:         if (currentToken.lexeme != "==" && currentToken.lexeme != "^=" && currentToken.lexeme != ">" && currentToken.lex
eme != "<" && currentToken.lexeme != "=>" && currentToken.lexeme != "=<")
  708:         {
  709:                 throw SyntaxError("Expected relational operator", currentToken.lineNumber);
  710:         }
  711:
  712:         this->savedOp = new std::string(currentToken.lexeme);
  713:
  714:         if (print)
  715:         {
  716:                 output << "\t<Relop> -> " << currentToken.lexeme << std::endl;
  717:         }
  718: }
  719:
  720: void SyntaxAnalyzer::Empty()
  721: {
  722:         if (print)
  723:         {
  724:                 output << "\t<Empty> -> Îµ" << std::endl;
  725:         }
  726: }
  727:
  728: void SyntaxAnalyzer::Body()
  729: {
  730:         if (print)
  731:         {
  732:                 output << "\t<Body> -> { <Statement List> }" << std::endl;
  733:         }
  734:
  735:         if (currentToken.lexeme != "{")
  736:         {
  737:                 throw SyntaxError("Expected '{'", currentToken.lineNumber);
  738:         }
  739:
  740:         getNextToken();
  741:
  742:         StatementList();
  743:
  744:         if (currentToken.lexeme != "}")
  745:         {
  746:                 throw SyntaxError("Expected '}'", currentToken.lineNumber);
  747:         }
  748:
  749:         getNextToken();
  750: }
  751:
  752: void SyntaxAnalyzer::FunctionDefinitions()
  753: {
  754:         if (print)
  755:         {
  756:                 output << "\t<Function Definitions> -> <Function> | <Function> <Function Definitions>" << std::endl;
  757:         }
  758:
```

```
759:         Function();
760:
761:         if (currentToken.lexeme == "function")
762:         {
763:                 getNextToken();
764:                 FunctionDefinitions();
765:         }
766: }
767:
768: void SyntaxAnalyzer::Print()
769: {
770:         if (print)
771:         {
772:                 output << "\t<Print> -> put ( <Expression> );" << std::endl;
773:         }
774:
775:         if (currentToken.lexeme != "(")
776:         {
777:                 throw SyntaxError("Expected '('", currentToken.lineNumber);
778:         }
779:
780:         getNextToken();
781:         Expression();
782:
783:         if (currentToken.lexeme != ")")
784:         {
785:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
786:         }
787:         getNextToken();
788:
789:         if (currentToken.lexeme != ";")
790:         {
791:                 throw SyntaxError("Expected ';'", currentToken.lineNumber);
792:         }
793:
794:         symbolTable.gen_instr("STDOUT", NIL);
795:
796:         getNextToken();
797: }
798:
799: void SyntaxAnalyzer::Scan()
800: {
801:         if (print)
802:         {
803:                 output << "\t<Scan> -> get ( <IDs> );" << std::endl;
804:         }
805:
806:         if (currentToken.lexeme != "(")
807:         {
808:                 throw SyntaxError("Expected '('", currentToken.lineNumber);
809:         }
810:
811:         getNextToken();
812:
813:         symbolTable.gen_instr("STDIN", NIL);
814:         int addr = symbolTable.get_address(currentToken);
815:         symbolTable.gen_instr("POPM", addr);
816:
817:         IDs();
818:
819:         if (currentToken.lexeme != ")")
820:         {
821:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
822:         }
823:
824:         getNextToken();
825:         if (currentToken.lexeme != ";")
826:         {
827:                 throw SyntaxError("Expected ';'", currentToken.lineNumber);
828:         }
829:
830:         getNextToken();
831: }
832:
833: void SyntaxAnalyzer::TermPrime()
834: {
835:         if (print)
836:         {
837:                 output << "\t<TermPrime> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>" << std::endl;
838:         }
839:
840:         if (currentToken.lexeme == "*" | currentToken.lexeme == "/")
841:         {
842:                 std::string op = currentToken.lexeme;
843:
844:                 getNextToken();
845:
846:                 Factor();
847:
848:                 if (op == "*")
849:                 {
850:                         symbolTable.gen_instr("MUL", NIL);
851:                 }
852:                 else
853:                 {
854:                         symbolTable.gen_instr("DIV", NIL);
```

```
  855:                 }
  856:
  857:                 TermPrime();
  858:         }
  859: }
  860:
  861: /**
  862:  * Attempt to syntactically analyze a list of
  863:  * Lexer tokens
  864:  */
  865: void SyntaxAnalyzer::Analyze()
  866: {
  867:         Rat18F();
  868:         output << "Syntax Analysis Successful." << std::endl << std::endl;
  869: }
  870:
  871: void SyntaxAnalyzer::OptParameterList()
  872: {
  873:         if (print)
  874:         {
  875:                 output << "\t<Opt Parameter List> -> <Parameter List> | <Empty>" << std::endl;
  876:         }
  877:
  878:         if (currentToken.lexeme == ")")
  879:         {
  880:                 Empty();
  881:         }
  882:         else if (currentToken.token == "Identifier")
  883:         {
  884:                 ParameterList();
  885:         }
  886:         else
  887:         {
  888:                 throw SyntaxError("Expected ')' or identifier", currentToken.lineNumber);
  889:         }
  890: }
  891:
  892: void SyntaxAnalyzer::ParameterList()
  893: {
  894:         if (print)
  895:         {
  896:                 output << "\t<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>" << std::endl;
  897:         }
  898:
  899:         Parameter();
  900:
  901:         getNextToken();
  902:
  903:         if (currentToken.lexeme == ",")
  904:         {
  905:                 getNextToken();
  906:                 ParameterList();
  907:         }
  908: }
  909:
  910: void SyntaxAnalyzer::While()
  911: {
  912:         if (print)
  913:         {
  914:                 output << "\t<While> -> while ( <Condition> )  <Statement>" << std::endl;
  915:         }
  916:
  917:         int addr = symbolTable.get_instr_address();
  918:         symbolTable.gen_instr("LABEL", NIL);
  919:
  920:         if (currentToken.lexeme != "(")
  921:         {
  922:                 throw SyntaxError("Expected '('", currentToken.lineNumber);
  923:         }
  924:         getNextToken();
  925:
  926:         Condition();
  927:
  928:         if (currentToken.lexeme != ")")
  929:         {
  930:                 throw SyntaxError("Expected ')'", currentToken.lineNumber);
  931:         }
  932:         getNextToken();
  933:         Statement();
  934:
  935:         if (currentToken.lexeme != "whileend")
  936:         {
  937:                 throw SyntaxError("Expected 'whileend' keyword", currentToken.lineNumber);
  938:         }
  939:         symbolTable.gen_instr("JUMP", addr);
  940:         symbolTable.back_patch(symbolTable.get_instr_address());
  941:
  942:         getNextToken();
  943: }
  944:
  945: void SyntaxAnalyzer::printCurrentToken()
  946: {
  947:         output << std::left << std::endl
  948:                 << std::setw(8) << "Token:" << std::setw(16) << currentToken.token << std::setw(8) << "Lexeme:" << curre
ntToken.lexeme << std::endl
  949:                 << std::endl;
```

```
950: }
951:
952: SyntaxError::SyntaxError(std::string message, int lineNumber)
953: {
954:         this->message = message;
955:         this->lineNumber = lineNumber;
956: }
957:
958: SyntaxError::~SyntaxError() {}
959:
960: std::string SyntaxError::getMessage() const
961: {
962:         return (this->message + " Line: " + std::to_string(this->lineNumber));
963: }
964:
965: std::string SyntaxAnalyzer::PrintAll()
966: {
967:         std::ostringstream out;
968:         out << this->symbolTable.list();
969:         out << std::endl;
970:         out << this->symbolTable.list_instr();
971:         if (this->errCount > 0)
972:         {
973:                 out << std::endl;
974:                 out << errCount << " ERROR" << ((errCount > 1) ? "S" : "");
975:                 out << " FOUND" << std::endl;
976:                 out << std::setfill('-') << std::setw(15) << '-' << std::setfill(' ') << std::endl;
977:                 out << err.str();
978:         }
979:         else
980:         {
981:                 out << std::endl << "3AC Code Generated Successfully!" << std::endl;
982:         }
983:         out << std::endl;
984:
985:         return out.str();
986: }
```