

```

1: #include "SyntaxAnalyzer.h"
2:
3: SyntaxAnalyzer::SyntaxAnalyzer(const std::vector<Lexer::Token> &tokens, std::ofstream &output, bool print) : tokens(tokens), 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:     this->assign = false;
10: }
11:
12: SyntaxAnalyzer::~SyntaxAnalyzer()
13: {
14: }
15:
16: void SyntaxAnalyzer::error(ErrorType errorType, int lineNumber, std::string expected)
17: {
18:     errCount++;
19:     err << "[ERR] (Line " << lineNumber << " ) ";
20:     switch (errorType)
21:     {
22:     case TYPE_MISMATCH:
23:     {
24:         err << "TYPE MISMATCH";
25:         if (expected != "")
26:         {
27:             err << ". Expected \"" << expected << "\"";
28:         }
29:         break;
30:     }
31:     case DUPLICATE_SYMBOL:
32:     {
33:         err << "DUPLICATE SYMBOL";
34:         if (expected != "")
35:         {
36:             err << " \"" << expected << "\"";
37:         }
38:         break;
39:     }
40:     case UNDECLARED_VARIABLE:
41:     {
42:         err << "UNDECLARED VARIABLE";
43:         if (expected != "")
44:         {
45:             err << " \"" << expected << "\"";
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:         {
95:             getNextToken();
96:             OptDeclarationList();
97:             StatementList();
98:         }
99:
100:     if (currentToken.lexeme != "$$")
101:     {
102:         throw SyntaxError("Expected '$$'.", currentToken.lineNumber);
103:     }
104: }
105:
106: void SyntaxAnalyzer::Parameter()
107: {
108:     if (print)
109:     {
110:         output << "\t<Parameter> -> <IDs> : <Qualifier>" << std::endl;
111:     }
112:
113:     IDs();
114:
115:     if (currentToken.lexeme != ":")
116:     {
117:         throw SyntaxError("Expected ':'", currentToken.lineNumber);
118:     }
119:
120:     getNextToken();
121:     Qualifier();
122: }
123:
124: void SyntaxAnalyzer::Function()
125: {
126:     if (print)
127:     {
128:         output << "\t<Function> -> function <Identifier> ( <Opt Parameter List> ) <Opt Declaration List> <Body>"
" << std::endl;
129:     }
130:
131:     Identifier();
132:
133:     getNextToken();
134:     if (currentToken.lexeme != "(")
135:     {
136:         throw SyntaxError("Expected '(',", currentToken.lineNumber);
137:     }
138:
139:     getNextToken();
140:
141:     OptParameterList();
142:
143:     if (currentToken.lexeme != ")")
144:     {
145:         throw SyntaxError("Expected ')'", currentToken.lineNumber);
146:     }
147:
148:     getNextToken();
149:     OptDeclarationList();
150:     Body();
151: }
152:
153: void SyntaxAnalyzer::OptFunctionDefinitions()
154: {
155:     if (print)
156:     {
157:         output << "\t<Opt Function Definitions> -> <Function Definitions> | <Empty>" << std::endl;
158:     }
159:
160:     if (currentToken.lexeme == "function")
161:     {
162:         getNextToken();
163:         FunctionDefinitions();
164:     }
165:     else
166:     {
167:         Empty();
168:     }
169: }
170:
171: void SyntaxAnalyzer::OptDeclarationList()
172: {
173:     if (print)
174:     {
175:         output << "\t<Opt Declaration List> -> <Declaration List> | <Empty>" << std::endl;
176:     }
177:
178:     if (currentToken.lexeme == "real" | currentToken.lexeme == "boolean" | currentToken.lexeme == "int")
179:     {
180:         DeclarationList();
181:     }
182:     else
183:     {
184:         Empty();
185:     }
186: }
187:
188: void SyntaxAnalyzer::DeclarationList()
```

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

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

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

```

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

```

```
568:         }
569:
570:         Real();
571:         getNextToken();
572:     }
573:     else if (currentToken.lexeme == "true")
574:     {
575:         // Error TYPE MISMATCH
576:         if (!symbolTable.typestack_empty() && symbolTable.top_typestack() != "boolean")
577:         {
578:             error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
579:         }
580:
581:         if (print)
582:         {
583:             output << "\ttrue" << std::endl;
584:         }
585:
586:         symbolTable.gen_instr("PUSHI", 1);
587:         getNextToken();
588:     }
589:     else if (currentToken.lexeme == "false")
590:     {
591:         // Error TYPE MISMATCH
592:         if (!symbolTable.typestack_empty() && symbolTable.top_typestack() != "boolean")
593:         {
594:             error(TYPE_MISMATCH, currentToken.lineNumber, symbolTable.top_typestack());
595:         }
596:
597:         if (print)
598:         {
599:             output << "\tfalse" << std::endl;
600:         }
601:         symbolTable.gen_instr("PUSHI", 0);
602:         getNextToken();
603:     }
604: }
605:
606: void SyntaxAnalyzer::Integer()
607: {
608:     if (print)
609:     {
610:         output << "\t<Integer>" << std::endl;
611:     }
612: }
613:
614: void SyntaxAnalyzer::Real()
615: {
616:     if (print)
617:     {
618:         output << "\t<Real>" << std::endl;
619:     }
620: }
621:
622: void SyntaxAnalyzer::Return()
623: {
624:     if (print)
625:     {
626:         output << "\t<Return> -> return; | return <Expression>;" << std::endl;
627:     }
628:
629:     if (currentToken.lexeme != ";")
630:     {
631:         Expression();
632:     }
633:     getNextToken();
634: }
635:
636: void SyntaxAnalyzer::If()
637: {
638:     if (print)
639:     {
640:         output << "\t<If> -> if ( <Condition> ) <Statement> endif | if ( <Condition> ) <Statement> else <Stateme
nt> endif" << std::endl;
641:     }
642:
643:     if (currentToken.lexeme != "(")
644:     {
645:         throw SyntaxError("Expected '(', currentToken.lineNumber);
646:     }
647:
648:     getNextToken();
649:
650:     Condition();
651:
652:     if (currentToken.lexeme != ")")
653:     {
654:         throw SyntaxError("Expected ')', currentToken.lineNumber);
655:     }
656:
657:     getNextToken();
658:
659:     Statement();
660:
661:     if (currentToken.lexeme == "else")
662:     {
```

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



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

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

```
950:         if (currentToken.lexeme != ")")
951:         {
952:             throw SyntaxError("Expected ')'", currentToken.lineNumber);
953:         }
954:         getNextToken();
955:         Statement();
956:
957:         if (currentToken.lexeme != "whileend")
958:         {
959:             throw SyntaxError("Expected 'whileend' keyword", currentToken.lineNumber);
960:         }
961:         symbolTable.gen_instr("JUMP", addr);
962:         symbolTable.back_patch(symbolTable.get_instr_address());
963:
964:         getNextToken();
965:     }
966:
967: void SyntaxAnalyzer::printCurrentToken()
968: {
969:     output << std::left << std::endl
970:             << std::setw(8) << "Token:" << std::setw(16) << currentToken.token << std::setw(8) << "Lexeme:" << cu
rrrentToken.lexeme << std::endl
971:             << std::endl;
972: }
973:
974: SyntaxError::SyntaxError(std::string message, int lineNumber)
975: {
976:     this->message = message;
977:     this->lineNumber = lineNumber;
978: }
979:
980: SyntaxError::~SyntaxError() {}
981:
982: std::string SyntaxError::getMessage() const
983: {
984:     return (this->message + " Line: " + std::to_string(this->lineNumber));
985: }
986:
987: std::string SyntaxAnalyzer::PrintAll()
988: {
989:     std::ostringstream out;
990:     out << this->symbolTable.list();
991:     out << std::endl;
992:     out << this->symbolTable.list_instr();
993:     if (this->errCount > 0)
994:     {
995:         out << std::endl;
996:         out << errCount << " ERROR" << ((errCount > 1) ? "S" : "");
997:         out << " FOUND" << std::endl;
998:         out << std::setfill('-') << std::setw(15) << '-' << std::setfill(' ') << std::endl;
999:         out << err.str();
1000:     }
1001:     else
1002:     {
1003:         out << std::endl
1004:             << "3AC Code Generated Successfully!" << std::endl;
1005:     }
1006:     out << std::endl;
1007:
1008:     return out.str();
1009: }
```