COVER PAGE

CS323 Programming Assignments

Andres Imperial

Assignment 1: Syntax Analyzer/Parser

Due: November 6

Turned in: November 6

Executable FileName [Parsley]

Lab Room: Titan online server

OS: Linux Mint 18 ‘Sarah’ 64bit

GRADE:

COMMENTS:

CS323 Assignment Documentation
<should consists of about 2-3 pages>

1. **Problem Statement**
   Construct a syntax analyzer using a Recursive-Decent Parser(RDP)

2. **How to use your program**
   Parsley is the name of the executable file. In order to run a lexical analysis and parse a file for syntax analysis according to the RAT16F language provide the file name as an argument in the command line when running *Parsley.* Proper format follows:

   ./Parsley *yourfile.txt*

   After execution *Parsley* will create a new file in the current directory labeled:

   parser*_yourfile.txt*

   This file will then contain all production rules utilized when parsing *yourfile.txt.*

3. **Design of your program**
   I choose to go with the Recursive-Decent Parser for its simplicity. I also chose to utilize a dynamic 2D array to house the tokens and lexemes.

   All on the production rules follow and have been modified for left recursion:

<Rat16F> ::= $$ <Opt Function Definitions>
$$ <Opt Declaration List> <Statement List> $$
<Opt Function Definitions> ::= <Function Definitions> | <Empty>
<Function Definitions> ::= <Function> | <Function> <Function Definitions>
<Function> ::= function <Identifier> [<Opt Parameter List>] <Opt Declaration List>
<Body>
<Opt Parameter List> ::= <Parameter List> | <Empty>
<Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>
<Parameter> ::= <IDs > : <Qualifier>
<Qualifier> ::= integer | boolean | real
<Body> ::= { < Statement List> }
<Opt Declaration List> ::= <Declaration List> | <Empty>
<Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> ::= <Qualifier > <IDs>
<IDs> ::= <Identifier> | <Identifier>, <IDs>
<Statement List> ::= <Statement> | <Statement> <Statement List>
<Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> |
<While>
<Compound> ::= { <Statement List> }
<Assign> ::= <Identifier> := <Expression> ;
<If> ::= if (<Condition>) <Statement> endif |

if (<Condition>) <Statement> else <Statement> endif
<Return> ::= return ; | return <Expression> ;
<Write> ::= print (<Expression>);
<Read> ::= read (<IDs>);
<While> ::= while (<Condition>) <Statement>
<Condition> ::= <Expression> <Relop> <Expression>
<Relop> ::= = | /= | > | < | => | <=
<Factor> ::= - <Primary> | <Primary>
<Primary> ::= <Identifier> | <Integer> | <Identifier> [<IDs>] |
(<Expression>) | <Real> | true | false
<Empty> ::= epsilon

Left Recursive:
<Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>

<Term> ::= <Term> * <Factor> | <Term> / <Factor> | <Factor>

Rewriting it without left recursion:
<Expression> ::= <Term> <Expression Prime>
<Expression Prime> ::= +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

<Term> ::= <Factor> <Term Prime>
<Term Prime> ::= * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon

4. **Any Limitation**
The only limitations for the program are that once an error is encountered the program does not recover and instead exits.

5. **Any shortcomings**
I was not able to get the line number in the source program file, instead when an error occurs the line number refers the line number in the lexer file.

```
// ------------------------------------------------------------------------
// Andres Imperial
// CSCP 323 mw 11:30am
// Assignment 2: Parser
//
// file: parser.h
// ------------------------------------------------------------------------

#ifndef PARSER_H
#define PARSER_H
#include <iostream>
#include <stdlib.h>
#include <cstring>
#include <fstream>
#include "parser_functs.h"
using namespace std;
const int token = 0;
const int lexeme = 1;
// Global variables
string** lexerArr;
int arrIndex = 1;
ofstream parserFile;

bool Parser(string fileName, string sourceName, int lineCount)
{

  ifstream lexerFile;

  lexerArr = new string*[lineCount + 1];
  bool compile = true;

  // Open file for reading
  lexerFile.open(fileName.c_str());
  parserFile.open((string("parser_") + sourceName).c_str());

  // If file opened properly
  if(!lexerFile.is_open()){
    cout << "Error file -- " << fileName << " -- could not be opened!\n";
    return 0;
  }
  if(!parserFile.is_open()){
    cout << "Error -- parser.txt could not be opened.\n";
    return 0;
  }

  for(int i = 0; !lexerFile.eof(); ++i){
    lexerArr[i] = new string[2];
    // Load array
    lexerFile >> lexerArr[i][token]; // Token
    lexerFile >> lexerArr[i][lexeme]; // Lexeme
  }

  // Start the Parsing at the root
  if(!Rat16F()){
    // Failed Parsing
    cout << "Error unable to parse file!\n";
    compile = false;
  }

  // Delete dynamic memory
  for(int i = 0; i < lineCount + 1; ++i){
    delete[] lexerArr[i];
  }
  delete[] lexerArr;

  lexerFile.close();
  parserFile.close();
```

```
68
69    return compile;
70
71  } // End of Parser()
72
73
74  // ----- ErrorMsg =-------------------------------------------------------
75  // -----------------------------------------------------------------------
76  void ErrorMsg(string msg)
77  {
78    // Print out error message with line number, given token and lexeme, and
79    // expected lexeme or token
80
81    cout << "Error on line: " << arrIndex << " -- expected " << msg
82      << " instead received lexeme: " << lexerArr[arrIndex][lexeme]
83      << " token type: " << lexerArr[arrIndex][token] << endl;
84
85    // Exit program
86    exit(0);
87
88  } // End of ErrorMsg()
89
90
91  // ----- lexeme_is -------------------------------------------------------
92  // -----------------------------------------------------------------------
93  bool lexeme_is(string target)
94  {
95
96    // Create compile flag
97    bool compile = true;
98
99    if(lexerArr[arrIndex][lexeme] == target){
100     ++arrIndex;
101
102     parserFile << "\nToken: " << lexerArr[arrIndex][token] << "\tLexeme: "
103     << lexerArr[arrIndex][lexeme] << endl;
104   }
105   else{
106     // Lexemes did not match
107     compile = false;
108   }
109
110   return compile;
111
112 } // End of lexeme_is()
113
114
115 // ----- token_is --------------------------------------------------------
116 // -----------------------------------------------------------------------
117 bool token_is(string target)
118 {
119
120   // Create compile flag
121   bool compile = true;
122
123   if(lexerArr[arrIndex][token] == target){
124     ++arrIndex;
125
126     parserFile << "\nToken: " << lexerArr[arrIndex][token] << "\tLexeme: "
127     << lexerArr[arrIndex][lexeme] << endl;
128   }
129   else{
130     // Tokens did not match
131     compile = false;
132   }
133
134   return compile;
```

```
135
136  } // End of token_is()
137
138
139  // ----- Rat16F -----------------------------------------------------------
140  // ------------------------------------------------------------------------
141  bool Rat16F(void)
142  {
143
144    // First Production Rule
145    parserFile << "Token: " << lexerArr[arrIndex][token] << "\tLexeme: "
146      << lexerArr[arrIndex][lexeme] << endl;
147    parserFile << "<Rat16F> -> $$ <Opt Function Definitions>\n"
148          "$$ <Opt Declaration List> <Statement List> $$\n";
149
150    // Create compile flag
151    bool compile = false;
152    // Program must start with $$ marker
153    if(lexeme_is("$$")){
154      if(Opt_Funct_Def()){
155        if(lexeme_is("$$")){
156          if(Opt_Declar_List()){
157            if(Statement_List()){
158              if(lexeme_is("$$")){
159                // File was syntactically correct.
160                compile = true;
161              }
162              else{
163                ErrorMsg("$$");
164              }
165            }
166            else{
167              ErrorMsg("<Statement_List>");
168            }
169          }
170        }
171        else{
172          ErrorMsg("$$");
173        }
174      }
175    }
176    else{
177      ErrorMsg("$$");
178    }
179
180    return compile;
181
182  } // End of Rat16F()
183
184
185  // ----- Opt_Funct_Def ----------------------------------------------------
186  // ------------------------------------------------------------------------
187  bool Opt_Funct_Def(void)
188  {
189
190    // Production Rule
191    parserFile << "<Opt Function Definitions> -> <Function Definitions> | <Empty>\n";
192
193    // Create compile flag
194    bool compile = true;
195
196    if (Funct_Def()){
197      return compile;
198    }
199
200    // It was empty, but acceptable
201    return compile;
```

```
202
203  } // End of Opt_Funct_Def()
204
205
206  // ----- Opt_Declar_List ------------------------------------------------------
207  // ----------------------------------------------------------------------------
208  bool Opt_Declar_List(void)
209  {
210
211    // Production Rule
212    parserFile << "<Opt Declaration List> -> <Declaration List> | <Empty>\n";
213
214    // Create compile flag
215    bool compile = true;
216
217    if (Declar_List()){
218      return compile;
219    }
220
221    // It was empty, but acceptable
222    return compile;
223
224  } // End of Opt_Declar_List()
225
226
227  // ----- Statement_List -------------------------------------------------------
228  // ----------------------------------------------------------------------------
229  bool Statement_List(void)
230  {
231
232    // Production Rule
233    parserFile << "<Statement List> -> <Statement> | <Statement> <Statement List>\n";
234
235    // Create compile flag
236    bool compile = true;
237
238    if(Statement()){
239      while(Statement());
240    }
241    else{
242      // Fail on Statement()
243      compile = false;
244    }
245
246    return compile;
247
248  } // End of Statement_List()
249
250
251  // ----- Funct_Def ------------------------------------------------------------
252  // ----------------------------------------------------------------------------
253  bool Funct_Def(void)
254  {
255
256    // Production Rule
257    parserFile << "<Function Definitions> -> <Function> | "
258        "<Function> <Function Definitions>\n";
259
260    // Create compile flag
261    bool compile = true;
262
263    if (Function()){
264      if (Funct_Def()){
265      }
266    }
267    else{
268      // Fail on Function()
```

```
269       compile = false;
270     }
271
272     return compile;
273
274   } // End of Funct_Def()
275
276
277   // ----- Function -----------------------------------------------------------
278   // --------------------------------------------------------------------------
279   bool Function(void)
280   {
281
282     // Production Rule
283     parserFile << "<Function> -> function  <Identifier> [ <Opt Parameter List> ] "
284         "<Opt Declaration List> <Body>\n";
285
286     // Create compile flag
287     bool compile = true;
288
289     // Function must start with keyword function
290     if (lexeme_is("function")){
291       if (Identifier()){
292         if (lexeme_is("[")){
293           if (Opt_Param_List()){
294             if (lexeme_is("]")){
295               if (Opt_Declar_List()){
296                 if (Body()){
297                 }
298                 else{
299                   // Failed on Body()
300                   compile = false;
301                 }
302               }
303               else{
304                 // Failed on Opt_Declar_List()
305                 compile = false;
306               }
307             }
308             else{
309               // Failed on lexeme_is("]")
310               ErrorMsg("]");
311               compile = false;
312             }
313           }
314           else{
315             // Failed on Opt_Param_List()
316             compile = false;
317           }
318         }
319         else{
320           // Failed on lexeme_is("[")
321           ErrorMsg("[");
322           compile = false;
323         }
324       }
325       else{
326         // Failed on Identifier()
327         compile = false;
328       }
329     }
330     else{
331       // Failed on lexeme_is("function")
332       compile = false;
333     }
334
335     return compile;
```

```
336
337   } // End of Function()
338
339
340   // ----- Identifier -------------------------------------------------------
341   // ------------------------------------------------------------------------
342   bool Identifier(void)
343   {
344
345     // Create compile flag
346     bool compile = true;
347
348     if(!token_is("identifier")){
349       // Failed on token_is identifer
350       compile = false;
351     }
352
353     return compile;
354
355   } // End of Identifier()
356
357
358   // ----- Opt_Param_List ---------------------------------------------------
359   // ------------------------------------------------------------------------
360   bool Opt_Param_List(void)
361   {
362
363     // Production Rule
364     parserFile << "<Opt Parameter List> ->  <Parameter List> | <Empty>\n";
365
366     // Create compile flag
367     bool compile = true;
368
369     if(Param_List()){
370       return compile;
371     }
372
373     // It was empty but acceptable.
374     return compile;
375
376   } // End of Opt_Param_List()
377
378
379   // ----- Body -------------------------------------------------------------
380   // ------------------------------------------------------------------------
381   bool Body(void)
382   {
383
384     // Production Rule
385     parserFile << "<Body> -> { <Statement List> }\n";
386
387     // Create compile flag
388     bool compile = true;
389
390     // Body must start with {
391     if(lexeme_is("{")){
392       if(Statement_List()){
393         if(lexeme_is("}")){
394           // used <Body>  ::=  {  < Statement List>  }
395         }
396         else{
397           // failed on lexeme_is("}")
398           ErrorMsg("}");
399           compile = false;
400         }
401       }
402       else{
```

```
403         // failed on Statement_List()
404         compile = false;
405       }
406     }
407     else{
408       // failed on lexeme_is("{")
409       ErrorMsg("{");
410       compile = false;
411     }
412
413     return compile;
414
415   } // End of Body()
416
417
418   // ----- Param_List ----------------------------------------------------
419   // ---------------------------------------------------------------------
420   bool Param_List(void)
421   {
422
423     // Production Rule
424     parserFile << "<Parameter List> -> <Parameter> | <Parameter> , "
425         "<Parameter List>\n";
426
427     // Create compile flag
428     bool compile = true;
429
430       if(Parameter()){
431       while(lexeme_is(",")){
432         Parameter();
433       }
434       }
435       else{
436           // Failed on Parameter()
437           compile = false;
438       }
439
440       return compile;
441
442   } // End of Param_List()
443
444
445   // ----- Parameter -----------------------------------------------------
446   // ---------------------------------------------------------------------
447   bool Parameter(void)
448   {
449
450     // Production Rule
451     parserFile << "<Parameter> -> <IDs> : <Qualifier>\n";
452
453     // Create compile flag
454     bool compile = true;
455
456       if(IDs()){
457           if(lexeme_is(":")){
458               if(Qualifier()){
459                   // <Parameter> ::=  <IDs > : <Qualifier>
460               }
461               else{
462                   // Failed on Qualifier()
463                   compile = false;
464               }
465           }
466           else{
467               // Failed on lexeme_is(":")
468         ErrorMsg(":");
469               compile = false;
```

```
470         }
471     }
472     else{
473         // Failed on IDs()
474         compile = false;
475     }
476 
477     return compile;
478 
479 } // End of Parameter()
480 
481 // ----- IDs ------------------------------------------------------------------
482 // ----------------------------------------------------------------------------
483 bool IDs(void)
484 {
485 
486   // Production Rule
487   parserFile << "<IDs> -> <Identifier> | <Identifier>, <IDs>\n";
488 
489   // Create compile flag
490   bool compile = true;
491 
492     if(Identifier()){
493     while(lexeme_is(",")){
494       if(Identifier()){
495       }
496     }
497     }
498     else{
499         // Failed on Identifier()
500     ErrorMsg("<identifier>");
501         compile = false;
502     }
503 
504     return compile;
505 
506 } // End of IDs()
507 
508 
509 // ----- Qualifier ------------------------------------------------------------
510 // ----------------------------------------------------------------------------
511 bool Qualifier(void)
512 {
513 
514   // Production Rule
515   parserFile << "<Qualifier> -> integer | boolean | real\n";
516 
517   // Create compile flag
518   bool compile = true;
519 
520     if(lexeme_is("integer") | lexeme_is("boolean") | lexeme_is("real")){
521         // <Qualifier> ::= integer   |  boolean  |  real
522     }
523     else{
524         // Failed on lexeme_is() expected.....
525         compile = false;
526     }
527 
528     return compile;
529 
530 } // End of Qualifier()
531 
532 
533 // ----- Declar_List ----------------------------------------------------------
534 // ----------------------------------------------------------------------------
535 bool Declar_List(void)
536 {
```

```
537
538     // Production Rule
539     parserFile << "<Declaration List> -> <Declaration> ; | <Declaration> ; "
540         "<Declaration List>\n";
541
542     // Create compile flag
543     bool compile = true;
544
545       if(Declaration()){
546           if(lexeme_is(";")){
547               Declar_List();
548           }
549           else{
550               // Failed on lexeme_is(";")
551         ErrorMsg(";");
552               compile = false;
553           }
554       }
555       else{
556           // Failed on Declaration()
557           compile = false;
558       }
559
560       return compile;
561
562   } // End of Declar_List()
563
564
565   // ----- Declaration --------------------------------------------------------
566   // --------------------------------------------------------------------------
567   bool Declaration(void)
568   {
569
570     // Production Rule
571     parserFile << "<Declaration> -> <Qualifier> <IDs>\n";
572     // Create compile flag
573     bool compile = true;
574
575       if(Qualifier()){
576           if(IDs()){
577               // <Declaration> ::=  <Qualifier > <IDs>
578           }
579           else{
580               // Failed on IDs()
581               compile = false;
582           }
583       }
584       else{
585           // Failed on Qualifier()
586           compile = false;
587       }
588
589       return compile;
590
591   } // End of Declaration()
592
593
594   // ----- Statement ----------------------------------------------------------
595   // --------------------------------------------------------------------------
596   bool Statement(void)
597   {
598
599     // Production Rule
600     parserFile << "<Statement> -> <Compound> | <Assign> | <If> |  <Return> | "
601           "<Write> | <Read> | <While>\n";
602
603     // Create compile flag
```

```
604     bool compile = true;
605
606       if(Compound() | Assign() | If() | Return() | Write() | Read() | While()){
607           // <Statement> ::=  <Compound> | <Assign> | <If> |  <Return> | <Write>
608           // | <Read> | <While>
609       }
610       else{
611           // Failed on <Compound> | <Assign> | <If> |  <Return>
612           // | <Write> | <Read> | <While>
613           compile = false;
614       }
615
616       return compile;
617
618   } // End of Statement()
619
620
621   // ----- Compound ------------------------------------------------------------
622   // ---------------------------------------------------------------------------
623   bool Compound(void)
624   {
625
626     // Production Rule
627     parserFile << "<Compound> -> { <Statement List> }\n";
628
629     // Create compile flag
630     bool compile = true;
631
632       // Must start with "{"
633       if(lexeme_is("{")){
634           if(Statement_List()){
635               if(lexeme_is("}")){
636                   // <Compound> ::= {  <Statement List>  }
637               }
638               else{
639                   // Failed on lexeme_is("}")
640           ErrorMsg("}");
641                   compile = false;
642               }
643           }
644           else{
645               // Failed on Statement_List()
646               compile = false;
647           }
648       }
649       else{
650           // Failed on lexeme_is("{")
651           compile = false;
652       }
653
654       return compile;
655
656   } // End of Compound()
657
658
659   // ----- Assign --------------------------------------------------------------
660   // ---------------------------------------------------------------------------
661   bool Assign(void)
662   {
663
664     // Production Rule
665     parserFile << "<Assign> -> <Identifier> := <Expression>;\n";
666
667     // Create compile flag
668     bool compile = true;
669
670       if(Identifier()){
```

```
671           if(lexeme_is(":=")){
672               if(Expression()){
673                   if(lexeme_is(";")){
674                       // <Assign> ::=   <Identifier> := <Expression> ;
675                   }
676                   else{
677                       // Failed on lexeme_is(";")
678             ErrorMsg(";");
679                       compile = false;
680                   }
681               }
682               else{
683                   // Failed on Expression()
684                   compile = false;
685               }
686           }
687           else{
688               // Failed on lexeme_is(":=")
689         ErrorMsg(":=");
690               compile = false;
691           }
692       }
693       else{
694          // Failed on Identifier()
695          compile = false;
696       }
697 
698       return compile;
699 
700   } // End of Assign()
701 
702 
703   // ----- If --------------------------------------------------------------------
704   // -----------------------------------------------------------------------------
705   bool If(void)
706   {
707 
708     // Production Rule
709     parserFile << "<If> -> if (<Condition>) <Statement> endif |\n"
710                "if (<Condition>) <Statement> else <Statement> endif\n";
711 
712     // Create compile flag
713     bool compile = true;
714 
715       // Must start with "if" keyword
716       if(lexeme_is("if")){
717           if(lexeme_is("(")){
718               if(Condition()){
719                   if(lexeme_is(")")){
720                       if(Statement()){
721                           if(lexeme_is("endif")){
722                               // <If> ::=     if  ( <Condition>  ) <Statement>
723                               // endif    |
724                           }
725                           else if(lexeme_is("else")){
726                               if(Statement()){
727                                   if(lexeme_is("endif")){
728                                       // <If> ::= if ( <Condition>  ) <Statement>
729                                       // else <Statement> endif
730                                   }
731                                   else{
732                                       // Failed on lexeme_is("endif")
733                     ErrorMsg("endif");
734                                       compile = false;
735                                   }
736                               }
737                               else{
```

```
                                    // Failed on Statement()
                                    compile = false;
                                }
                            }
                            else{
                                // Failed on lexeme_is() expected endif or else
                ErrorMsg("endif | else");
                                compile = false;
                            }
                        }
                        else{
                            // Failed on Statement()
                            compile = false;
                        }
                    }
                    else{
                        // Failed on lexeme_is(")")
            ErrorMsg(")");
                        compile = false;
                    }
                }
                else{
                    // Failed on Condition()
                    compile = false;
                }
            }
            else{
                // Failed on lexeme_is("(")
          ErrorMsg("(");
                compile = false;
            }
        }
        else{
            // Failed on lexeme_is("if")
            compile = false;
        }

        return compile;

    } // End of If()


    // ----- Return --------------------------------------------------------------
    // ---------------------------------------------------------------------------
    bool Return(void)
    {

      // Production Rule
      parserFile << "<Return> -> return ; | return <Expression> ;\n";

      // Create compile flag
      bool compile = true;

        if(lexeme_is("return")){
            if(lexeme_is(";")){
                // <Return> ::=  return ;
            }
            else if(Expression()){
                if(lexeme_is(";")){
                    // <Return> ::= return <Expression> ;
                }
                else{
                    // Failed on lexeme_is(";")
            ErrorMsg(";");
                    compile = false;
                }
            }
```

```
805           else{
806               // Failed, expected ";" or Expression
807         ErrorMsg("; | <Expression>");
808               compile = false;
809           }
810       }
811       else{
812           // Failed on lexeme_is("return")
813           compile = false;
814       }
815
816       return compile;
817
818   } // End of Return()
819
820
821   // ----- Write ---------------------------------------------------------------
822   // ---------------------------------------------------------------------------
823   bool Write(void)
824   {
825
826     // Production Rule
827     parserFile << "<Write> -> print (<Expression>);\n";
828
829     // Create compile flag
830     bool compile = true;
831
832       // Must start with keyword "print"
833       if(lexeme_is("print")){
834           if(lexeme_is("(")){
835               if(Expression()){
836                   if(lexeme_is(")")){
837                       if(lexeme_is(";")){
838                           // <Write> ::=   print ( <Expression>);
839                       }
840                       else{
841                           // Failed on lexeme_is(";")
842               ErrorMsg(";");
843                           compile = false;
844                       }
845                   }
846                   else{
847                       // Failed on lexeme_is(")")
848             ErrorMsg(")");
849                       compile = false;
850                   }
851               }
852               else{
853                   // Failed on Expression()
854                   compile = false;
855               }
856           }
857           else{
858               // Failed on lexeme_is("(")
859         ErrorMsg("(");
860               compile = false;
861           }
862       }
863       else{
864           // Failed on lexeme_is("print")
865           compile = false;
866       }
867
868       return compile;
869
870   } // End of Write()
871
```

```
872
873  // ----- Read ---------------------------------------------------------------
874  // --------------------------------------------------------------------------
875  bool Read(void)
876  {
877
878    // Production Rule
879    parserFile << "<Read> -> read (<IDs>);\n";
880
881    // Create compile flag
882    bool compile = true;
883
884      // Must start with keyword "read"
885      if(lexeme_is("read")){
886          if(lexeme_is("(")){
887              if(IDs()){
888                  if(lexeme_is(")")){
889                      if(lexeme_is(";")){
890                          // <Read> ::=    read ( <IDs> );
891                      }
892                      else{
893                          // Failed on lexeme_is(";")
894              ErrorMsg(";");
895                          compile = false;
896                      }
897                  }
898                  else{
899                      // Failed on lexeme_is(")")
900            ErrorMsg(")");
901                      compile = false;
902                  }
903              }
904              else{
905                  // Failed on IDs()
906                  compile = false;
907              }
908          }
909          else{
910              // Failed on lexeme_is("(")
911        ErrorMsg("(");
912              compile = false;
913          }
914      }
915      else{
916          // Failed on lexeme_is("read")
917          compile = false;
918      }
919
920      return compile;
921
922  } // End of Read()
923
924
925  // ----- While --------------------------------------------------------------
926  // --------------------------------------------------------------------------
927  bool While(void)
928  {
929
930    // Production Rule
931    parserFile << "<While> -> while (<Condition>) <Statement>\n";
932
933    // Create compile flag
934    bool compile = true;
935
936      // Must start with keyword "while"
937      if(lexeme_is("while")){
938          if(lexeme_is("(")){
```

```
939             if(Condition()){
940                 if(lexeme_is(")")){
941                     if(Statement()){
942                         // <While> ::= while ( <Condition>  )  <Statement>
943                     }
944                     else{
945                         // Failed on Statement()
946                         compile = false;
947                     }
948                 }
949                 else{
950                     // Failed on lexeme_is(")")
951           ErrorMsg(")");
952                     compile = false;
953                 }
954             }
955             else{
956                 // Failed on Condition()
957                 compile = false;
958             }
959         }
960         else{
961             // Failed on lexeme_is("(")
962       ErrorMsg("(");
963             compile = false;
964         }
965     }
966     else{
967         // Failed on lexeme_is("while")
968         compile = false;
969     }
970 
971     return compile;
972 
973 }// End of While()
974 
975 
976 // ----- Expression -------------------------------------------------------
977 // ------------------------------------------------------------------------
978 bool Expression(void)
979 {
980 
981   // Production Rule
982   parserFile << "<Expression> -> <Term> <Expression Prime>\n";
983 
984   // Create compile flag
985   bool compile = true;
986 
987     if(Term()){
988         if(Expression_Prime()){
989             // <Expression> ::= <Term> <Expression Prime>
990         }
991         else{
992             // Failed on Expression_Prime()
993             compile = false;
994         }
995     }
996     else{
997         // Failed on Term()
998         compile = false;
999     }
1000 
1001     return compile;
1002 
1003 } // End of Expression()
1004 
1005 
```

```
1006  // ----- Condition -------------------------------------------------------
1007  // -----------------------------------------------------------------------
1008  bool Condition(void)
1009  {
1010
1011    // Production Rule
1012    parserFile << "<Condition> -> <Expression> <Relop> <Expression>\n";
1013
1014    // Create compile flag
1015    bool compile = true;
1016
1017      if(Expression()){
1018          if(Relop()){
1019              if(Expression()){
1020                  // <Condition> ::= <Expression> <Relop> <Expression>
1021              }
1022              else{
1023                  // Failed on Expression()
1024                  compile = false;
1025              }
1026          }
1027          else{
1028              // Failed on Relop()
1029              compile = false;
1030          }
1031      }
1032      else{
1033          // Failed on Expression()
1034          compile = false;
1035      }
1036
1037      return compile;
1038
1039  } // End of Condition()
1040
1041
1042  // ----- Relop -----------------------------------------------------------
1043  // -----------------------------------------------------------------------
1044  bool Relop(void)
1045  {
1046
1047    // Production Rule
1048    parserFile << "<Relop> -> = | /= | > | < | => | <=\n";
1049
1050    // Create compile flag
1051    bool compile = true;
1052
1053      if(lexeme_is("=") | lexeme_is("/=") | lexeme_is(">") | lexeme_is("<") |
1054          lexeme_is("=>") | lexeme_is("<=")){
1055          //<Relop> ::=   = |  /=  |   >   | <   |  =>   | <=
1056      }
1057      else{
1058          // Failed on <Relop> ::=   = |  /=  |   >   | <   |  =>   | <=
1059      ErrorMsg("= | /= | > | < | => | <=");
1060          compile = false;
1061      }
1062
1063      return compile;
1064
1065  } // End of Relop()
1066
1067
1068  // ----- Expression_Prime ------------------------------------------------
1069  // -----------------------------------------------------------------------
1070  bool Expression_Prime(void)
1071  {
1072
```

```
1073    // Production Rule
1074    parserFile << "<Expression Prime> -> +<Term> <Expression Prime> | "
1075          "-<Term> <Expression Prime> | epsilon\n";
1076
1077    // Create compile flag
1078    bool compile = true;
1079
1080      if(lexeme_is("+") | lexeme_is("-")){
1081          if(Term()){
1082              if(Expression_Prime()){
1083                  //<Expression Prime> ::= +<Term> <Expression Prime> | -<Term>
1084                  //<Expression Prime> | epsilon
1085              }
1086              else{
1087                  // Failed on Expression_Prime()
1088                  compile = false;
1089              }
1090          }
1091          else{
1092              // Failed on Term()
1093              compile = false;
1094          }
1095      }
1096      else{
1097          // Was empty and moved to epsilon
1098          // <Expression Prime> ::= epsilon
1099      }
1100
1101      return compile;
1102
1103  } // End of Expression_Prime()
1104
1105
1106  // ----- Term ----------------------------------------------------------------
1107  // ---------------------------------------------------------------------------
1108  bool Term(void)
1109  {
1110
1111    // Production Rule
1112    parserFile << "<Term> -> <Factor> <Term Prime>\n";
1113
1114    // Create compile flag
1115    bool compile = true;
1116
1117      if(Factor()){
1118          if(Term_Prime()){
1119              // <Term>  ::= <Factor> <Term Prime>
1120          }
1121          else{
1122              // Failed on Term_Prime()
1123              compile = false;
1124          }
1125      }
1126      else{
1127          // Failed on Factor()
1128          compile = false;
1129      }
1130
1131      return compile;
1132
1133  } // End of Term()
1134
1135
1136  // ----- Term_Prime ----------------------------------------------------------
1137  // ---------------------------------------------------------------------------
1138  bool Term_Prime(void)
1139  {
```

```
1140
1141    // Production Rule
1142    parserFile << "<Term Prime> -> * <Factor> <Term Prime> | "
1143          "/ Factor <Term Prime> | epsilon\n";
1144
1145    // Create compile flag
1146    bool compile = true;
1147
1148      if(lexeme_is("*") | lexeme_is("/")){
1149          if(Factor()){
1150              if(Term_Prime()){
1151                  // <Term Prime> ::=  * <Factor> <Term Prime> | / Factor <Term
1152                  // Prime> | epsilon
1153              }
1154              else{
1155                  // Failed on Term_Prime()
1156                  compile = false;
1157              }
1158          }
1159          else{
1160              // Failed on Factor()
1161              compile = false;
1162          }
1163      }
1164      else{
1165          // It moved to epsilon
1166          // <Term Prime> ::= epsilon
1167      }
1168
1169      return compile;
1170
1171  } // End Term_Prime()
1172
1173
1174  // ----- Factor --------------------------------------------------------------
1175  // ---------------------------------------------------------------------------
1176  bool Factor(void)
1177  {
1178
1179    // Production Rule
1180    parserFile << "<Factor> -> - <Primary> | <Primary>\n";
1181
1182    // Create compile flag
1183    bool compile = true;
1184
1185      if(lexeme_is("-")){
1186          if(Primary()){
1187              // <Factor> ::= - <Primary>
1188          }
1189          else{
1190              // Failed on Primary()
1191              compile = false;
1192          }
1193      }
1194      else if(Primary()){
1195          // <Factor> ::= <Primary>
1196      }
1197      else{
1198          // Failed expected "-" or Primary
1199      ErrorMsg("- | <Primary>");
1200          compile = false;
1201      }
1202
1203      return compile;
1204
1205  } // Factor()
1206
```

```
1207
1208  // ----- Primary -------------------------------------------------------------
1209  // ---------------------------------------------------------------------------
1210  bool Primary(void)
1211  {
1212
1213    // Production Rule
1214    parserFile << "<Primary> -> <Identifier> | <Integer> | <Identifier> "
1215          "[<IDs>] | (<Expression>) | <Real> | true | false\n";
1216
1217    // Create compile flag
1218    bool compile = true;
1219
1220      if(Identifier()){
1221          if(lexeme_is("[")){
1222              if(IDs()){
1223                  if(lexeme_is("]")){
1224                      // <Primary> ::= <Identifier> [<IDs>]
1225                  }
1226                  else{
1227                      // Failed on lexeme_is("]")
1228            ErrorMsg("]");
1229                      compile = false;
1230                  }
1231              }
1232              else{
1233                  // Failed on IDs()
1234                  compile = false;
1235              }
1236          }
1237          else{
1238              // <Primary> ::= <Identifier>
1239          }
1240      }
1241      else if(token_is("integer")){
1242          // <Primary> ::= <Integer>
1243      }
1244      else if(lexeme_is("(")){
1245          if(Expression()){
1246              if(lexeme_is(")")){
1247                  // <Primary> ::= ( <Expression> )
1248              }
1249              else{
1250                  // Failed on lexeme_is(")")
1251          ErrorMsg(")");
1252                  compile = false;
1253              }
1254          }
1255          else{
1256              // Failed on Expression()
1257              compile = false;
1258          }
1259      }
1260      else if(token_is("real")){
1261          // <Primary> ::= <Real>
1262      }
1263      else if(lexeme_is("true") | lexeme_is("false")){
1264          // <Primary> ::= true | false
1265      }
1266      else{
1267          // Failed expected . . . .
1268      ErrorMsg("<Identifier> | <Integer> | <Identifier> [<IDs>] | "
1269          "<Expression> | <Real> | true | false");
1270          compile = false;
1271      }
1272
1273      return compile;
```

```
1274
1275  } // End of Primary()
1276
1277  #endif // End of parser.h
```

```
1  // ---------------------------------------------------------------------------
2  // Andres Imperial
3  // CSCP 323 mw 11:30am
4  // Assignment 2: Parser
5  //
6  // file: parser_functs.h
7  // ---------------------------------------------------------------------------
8
9  #ifndef PARSER_FUNCTS_H
10 #define PARSER_FUNCTS_H
11 #include <cstring>
12 using namespace std;
13
14 // Function Declarations
15
16 bool Parser(string fileName, int lineCount);
17 bool lexeme_is(string target);
18 bool token_is(string target);
19 bool Rat16F(void);
20 bool Opt_Funct_Def(void);
21 bool Opt_Declar_List(void);
22 bool Statement_List(void);
23 bool Funct_Def(void);
24 bool Function(void);
25 bool Identifier(void);
26 bool Opt_Param_List(void);
27 bool Body(void);
28 bool Param_List(void);
29 bool Parameter(void);
30 bool IDs(void);
31 bool Qualifier(void);
32 bool Declar_List(void);
33 bool Declaration(void);
34 bool Statement(void);
35 bool Compound(void);
36 bool Assign(void);
37 bool If(void);
38 bool Return(void);
39 bool Write(void);
40 bool Read(void);
41 bool While(void);
42 bool Expression(void);
43 bool Condition(void);
44 bool Relop(void);
45 bool Expression_Prime(void);
46 bool Term(void);
47 bool Term_Prime(void);
48 bool Factor(void);
49 bool Primary(void);
50
51 #endif // End of parser_functs.h
```

```
Token: separator  Lexeme: $$
<Rat16F> -> $$ <Opt Function Definitions>
$$ <Opt Declaration List> <Statement List> $$

Token: separator  Lexeme: $$
<Opt Function Definitions> -> <Function Definitions> | <Empty>
<Function Definitions> -> <Function> | <Function> <Function Definitions>
<Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>

Token: separator  Lexeme: {
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: identifier Lexeme: a
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: b
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: operator Lexeme: +
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: c
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false

Token: separator  Lexeme: ;
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: }
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: $$
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
```

```
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token:  Lexeme:
```

```
$$
$$
{
a := b + c;
}
$$
```

```
Token: separator  Lexeme: $$
<Rat16F> -> $$ <Opt Function Definitions>
$$ <Opt Declaration List> <Statement List> $$

Token: keyword  Lexeme: function
<Opt Function Definitions> -> <Function Definitions> | <Empty>
<Function Definitions> -> <Function> | <Function> <Function Definitions>
<Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>

Token: identifier Lexeme: Subtract

Token: separator  Lexeme: [

Token: identifier Lexeme: imVal
<Opt Parameter List> ->  <Parameter List> | <Empty>
<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: integer
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ]

Token: keyword  Lexeme: real
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: retVal92
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: separator  Lexeme: {
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Body> -> { <Statement List> }

Token: identifier Lexeme: imVal
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: imVal
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: -
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: (
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
```

```
66
67  Token: integer  Lexeme: 2
68  <Expression> -> <Term> <Expression Prime>
69  <Term> -> <Factor> <Term Prime>
70  <Factor> -> - <Primary> | <Primary>
71  <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
72
73  Token: operator Lexeme: *
74  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
75
76  Token: identifier Lexeme: imVal
77  <Factor> -> - <Primary> | <Primary>
78  <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
79
80  Token: separator  Lexeme: )
81  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
82  <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
83
84  Token: separator  Lexeme: ;
85  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
86  <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
87
88  Token: identifier Lexeme: retVal92
89  <If> -> if (<Condition>) <Statement> endif |
90  if (<Condition>) <Statement> else <Statement> endif
91  <Return> -> return ; | return <Expression> ;
92  <Write> -> print (<Expression>);
93  <Read> -> read (<IDs>);
94  <While> -> while (<Condition>) <Statement>
95  <Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
96  <Compound> -> { <Statement List> }
97  <Assign> -> <Identifier> := <Expression>;
98
99  Token: operator Lexeme: :=
100
101 Token: identifier Lexeme: imVal
102 <Expression> -> <Term> <Expression Prime>
103 <Term> -> <Factor> <Term Prime>
104 <Factor> -> - <Primary> | <Primary>
105 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
106
107 Token: separator  Lexeme: ;
108 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
109 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
110
111 Token: keyword  Lexeme: return
112 <If> -> if (<Condition>) <Statement> endif |
113 if (<Condition>) <Statement> else <Statement> endif
114 <Return> -> return ; | return <Expression> ;
115
116 Token: identifier Lexeme: retVal92
117 <Expression> -> <Term> <Expression Prime>
118 <Term> -> <Factor> <Term Prime>
119 <Factor> -> - <Primary> | <Primary>
120 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
121
122 Token: separator  Lexeme: ;
123 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
124 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
125
126 Token: separator  Lexeme: }
127 <Write> -> print (<Expression>);
128 <Read> -> read (<IDs>);
```

```
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: $$
<Function Definitions> -> <Function> | <Function> <Function Definitions>
<Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>

Token: keyword  Lexeme: integer
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ,

Token: identifier Lexeme: high_av

Token: separator  Lexeme: ;

Token: keyword  Lexeme: read
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ,

Token: identifier Lexeme: high_av

Token: separator  Lexeme: )

Token: separator  Lexeme: ;

Token: keyword  Lexeme: while
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
```

```
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: =
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
<Relop> -> = | /= | > | < | => | <=

Token: operator Lexeme: <

Token: identifier Lexeme: high_av
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: {
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: keyword  Lexeme: print
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);

Token: separator  Lexeme: (

Token: identifier Lexeme: low_av
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: ;

Token: keyword  Lexeme: print
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);

Token: separator  Lexeme: (

Token: identifier Lexeme: Subtract
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
```

```
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: [

Token: identifier Lexeme: low_av
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ]

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: ;

Token: separator  Lexeme: }
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: $$
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token:  Lexeme:
```

```
$$
function Subtract[imVal:integer]
  real retVal92;
{
  imVal := imVal - (2 * imVal);
  retVal92:=imVal;
  return retVal92;
}
$$
     integer   low_av,high_av;
     read(low_av,high_av);
     while (low_av =< high_av)
         {
    print(low_av);
              print(Subtract[low_av]);
         }
$$
```

```
Token: separator  Lexeme: $$
<Rat16F> -> $$ <Opt Function Definitions>
$$ <Opt Declaration List> <Statement List> $$

Token: keyword  Lexeme: function
<Opt Function Definitions> -> <Function Definitions> | <Empty>
<Function Definitions> -> <Function> | <Function> <Function Definitions>
<Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>

Token: identifier Lexeme: Hello

Token: separator  Lexeme: [

Token: identifier Lexeme: me
<Opt Parameter List> ->  <Parameter List> | <Empty>
<Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: real
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ,

Token: identifier Lexeme: alice
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: integer
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ,

Token: identifier Lexeme: bob
<Parameter> -> <IDs> : <Qualifier>
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: :

Token: keyword  Lexeme: boolean
<Qualifier> -> integer | boolean | real

Token: separator  Lexeme: ]

Token: keyword  Lexeme: boolean
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: modest
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: keyword  Lexeme: boolean
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: mouse
<IDs> -> <Identifier> | <Identifier>, <IDs>

```

```
Token: separator  Lexeme: ;

Token: keyword  Lexeme: real
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: pink
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: keyword  Lexeme: integer
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real

Token: identifier Lexeme: floyd
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: ;

Token: separator  Lexeme: {
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Body> -> { <Statement List> }

Token: identifier Lexeme: alice
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: me
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: ;
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: keyword  Lexeme: if
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif

Token: separator  Lexeme: (

Token: identifier Lexeme: pink
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: +
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: floyd
<Term> -> <Factor> <Term Prime>
```

```
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: >
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
<Relop> -> = | /= | > | < | => | <=

Token: identifier Lexeme: modest
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: /
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon

Token: identifier Lexeme: mouse
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: identifier Lexeme: me
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: alice
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: ;
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: keyword  Lexeme: else
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: identifier Lexeme: me
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;

Token: operator Lexeme: :=

Token: identifier Lexeme: modest
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
```

```
195
196  Token: operator Lexeme: +
197  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
198  <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
199
200  Token: identifier Lexeme: mouse
201  <Term> -> <Factor> <Term Prime>
202  <Factor> -> - <Primary> | <Primary>
203  <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
     false
204
205  Token: separator  Lexeme: ;
206  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
207  <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
208
209  Token: keyword  Lexeme: endif
210  <If> -> if (<Condition>) <Statement> endif |
211  if (<Condition>) <Statement> else <Statement> endif
212  <Return> -> return ; | return <Expression> ;
213  <Write> -> print (<Expression>);
214  <Read> -> read (<IDs>);
215  <While> -> while (<Condition>) <Statement>
216
217  Token: keyword  Lexeme: return
218  <Return> -> return ; | return <Expression> ;
219
220  Token: identifier Lexeme: me
221  <Expression> -> <Term> <Expression Prime>
222  <Term> -> <Factor> <Term Prime>
223  <Factor> -> - <Primary> | <Primary>
224  <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
     false
225
226  Token: separator  Lexeme: [
227
228  Token: identifier Lexeme: modest
229  <IDs> -> <Identifier> | <Identifier>, <IDs>
230
231  Token: separator  Lexeme: ,
232
233  Token: identifier Lexeme: alice
234
235  Token: separator  Lexeme: ]
236
237  Token: separator  Lexeme: ;
238  <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
239  <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
240
241  Token: separator  Lexeme: }
242  <Write> -> print (<Expression>);
243  <Read> -> read (<IDs>);
244  <While> -> while (<Condition>) <Statement>
245  <Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
246  <Compound> -> { <Statement List> }
247  <Assign> -> <Identifier> := <Expression>;
248  <If> -> if (<Condition>) <Statement> endif |
249  if (<Condition>) <Statement> else <Statement> endif
250  <Return> -> return ; | return <Expression> ;
251  <Write> -> print (<Expression>);
252  <Read> -> read (<IDs>);
253  <While> -> while (<Condition>) <Statement>
254
255  Token: separator  Lexeme: $$
256  <Function Definitions> -> <Function> | <Function> <Function Definitions>
257  <Function> -> function  <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>
258
259  Token: keyword  Lexeme: while
```

```
<Opt Declaration List> -> <Declaration List> | <Empty>
<Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> integer | boolean | real
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: (

Token: integer  Lexeme: 1
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: operator Lexeme: >
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
<Relop> -> = | /= | > | < | => | <=

Token: integer  Lexeme: 0
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false

Token: separator  Lexeme: )
<Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
<Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon

Token: separator  Lexeme: {
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }

Token: keyword  Lexeme: read
<Statement List> -> <Statement> | <Statement> <Statement List>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);

Token: separator  Lexeme: (

Token: identifier Lexeme: Hello
<IDs> -> <Identifier> | <Identifier>, <IDs>

Token: separator  Lexeme: )

Token: separator  Lexeme: ;

Token: separator  Lexeme: }
<While> -> while (<Condition>) <Statement>
```

```
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token: separator  Lexeme: $$
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>
<Statement> -> <Compound> | <Assign> | <If> |  <Return> | <Write> | <Read> | <While>
<Compound> -> { <Statement List> }
<Assign> -> <Identifier> := <Expression>;
<If> -> if (<Condition>) <Statement> endif |
if (<Condition>) <Statement> else <Statement> endif
<Return> -> return ; | return <Expression> ;
<Write> -> print (<Expression>);
<Read> -> read (<IDs>);
<While> -> while (<Condition>) <Statement>

Token:  Lexeme:
```

```
$$
function Hello[me:real, alice:integer, bob:boolean]

boolean modest; boolean mouse;
real pink;
integer floyd;
{
  alice := me;
  if(pink + floyd > modest/mouse)
    me := alice;
  else
    me := modest + mouse;
  endif

  return me[modest, alice];
}
$$
  while(1 > 0)
  {
    read(Hello);
  }
$$
```