

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.

```
1 // -----
2 // Andres Imperial
3 // CSCP 323 mw 11:30am
4 // Assignment 2: Parser
5 //
6 // file: parser.h
7 // -----
8
9 #ifndef PARSER_H
10 #define PARSER_H
11 #include <iostream>
12 #include <stdlib.h>
13 #include <cstring>
14 #include <fstream>
15 #include "parser_functs.h"
16 using namespace std;
17 const int token = 0;
18 const int lexeme = 1;
19 // Global variables
20 string** lexerArr;
21 int arrIndex = 1;
22 ofstream parserFile;
23
24 bool Parser(string fileName, string sourceName, int lineCount)
25 {
26     ifstream lexerFile;
27
28     lexerArr = new string*[lineCount + 1];
29     bool compile = true;
30
31     // Open file for reading
32     lexerFile.open(fileName.c_str());
33     parserFile.open((string("parser_") + sourceName).c_str());
34
35     // If file opened properly
36     if(!lexerFile.is_open()){
37         cout << "Error file -- " << fileName << " -- could not be opened!\n";
38         return 0;
39     }
40     if(!parserFile.is_open()){
41         cout << "Error -- parser.txt could not be opened.\n";
42         return 0;
43     }
44
45     for(int i = 0; !lexerFile.eof(); ++i){
46         lexerArr[i] = new string[2];
47         // Load array
48         lexerFile >> lexerArr[i][token]; // Token
49         lexerFile >> lexerArr[i][lexeme]; // Lexeme
50     }
51
52     // Start the Parsing at the root
53     if(!Rat16F()){
54         // Failed Parsing
55         cout << "Error unable to parse file!\n";
56         compile = false;
57     }
58
59     // Delete dynamic memory
60     for(int i = 0; i < lineCount + 1; ++i){
61         delete[] lexerArr[i];
62     }
63     delete[] lexerArr;
64
65     lexerFile.close();
66     parserFile.close();
67 }
```

```

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("$")){
159                             // File was syntactically correct.
160                             compile = true;
161                         }
162                     }
163                     else{
164                         ErrorMsg("$");
165                     }
166                 }
167                 else{
168                     ErrorMsg("<Statement_List>");
169                 }
170             }
171             else{
172                 ErrorMsg("$");
173             }
174         }
175     }
176     else{
177         ErrorMsg("$");
178     }
179
180     return compile;
181 } // End of Rat16F()
182
183
184 // ----- Opt_Funct_Def -----
185 // -----
186 bool Opt_Funct_Def(void)
187 {
188
189     // Production Rule
190     parserFile << "<Opt Function Definitions> -> <Function Definitions> | <Empty>\n";
191
192     // Create compile flag
193     bool compile = true;
194
195     if (Funct_Def()){
196         return compile;
197     }
198
199     // It was empty, but acceptable
200     return compile;
201

```

```

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                                 // Success
298                             }
299                             else{
300                                 // Failed on Body()
301                                 compile = false;
302                             }
303                         }
304                     }
305                     else{
306                         // Failed on Opt_Declar_List()
307                         compile = false;
308                     }
309                 }
310             }
311             else{
312                 // Failed on lexeme_is("[")
313                 ErrorMsg("[");
314                 compile = false;
315             }
316         }
317     }
318     else{
319         // Failed on Opt_Param_List()
320         compile = false;
321     }
322 }
323
324 }
325
326 else{
327     // Failed on Identifier()
328     compile = false;
329 }
330
331 else{
332     // Failed on lexeme_is("function")
333     compile = false;
334 }
335
336     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 identifier
350         compile = false;
351     }
352
353     return compile;
354 } // End of Identifier()
355
356
357 // ----- Opt_Param_List -----
358 // -----
359 bool Opt_Param_List(void)
360 {
361
362     // Production Rule
363     parserFile << "<Opt Parameter List> -> <Parameter List> | <Empty>\n";
364
365     // Create compile flag
366     bool compile = true;
367
368     if(Param_List()){
369         return compile;
370     }
371
372     // It was empty but acceptable.
373     return compile;
374 } // End of Opt_Param_List()
375
376
377 // ----- Body -----
378 // -----
379 bool Body(void)
380 {
381
382     // Production Rule
383     parserFile << "<Body> -> { <Statement List> }\n";
384
385     // Create compile flag
386     bool compile = true;
387
388     // Body must start with {
389     if(lexeme_is("{")){
390         if(Statement_List()){
391             if(lexeme_is("}")){
392                 // used <Body> ::= { <Statement List> }
393             }
394             else{
395                 // failed on lexeme_is("}")
396                 ErrorMsg("{}");
397                 compile = false;
398             }
399         }
400     }
401     else{
402

```

```

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     // Production Rule
486     parserFile << "<IDs> -> <Identifier> | <Identifier>, <IDs>\n";
487
488     // Create compile flag
489     bool compile = true;
490
491     if(Identifier()){
492         while(lexeme_is(",")){
493             if(Identifier()){
494             }
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 // ----- Qualifier -----
509 // -----
510 bool Qualifier(void)
511 {
512     // Production Rule
513     parserFile << "<Qualifier> -> integer | boolean | real\n";
514
515     // Create compile flag
516     bool compile = true;
517
518     if(lexeme_is("integer") | lexeme_is("boolean") | lexeme_is("real")){
519         // <Qualifier> ::= integer | boolean | real
520     }
521     else{
522         // Failed on lexeme_is() expected.....
523         compile = false;
524     }
525
526 return compile;
527
528 } // End of Qualifier()
529
530 // ----- Declar_List -----
531 // -----
532 bool Declar_List(void)
533 {

```

```

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 } // End of Statement()
618
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 } // End of Compound()
656
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 } // End of Assign()
700
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{

```

```

738             // Failed on Statement()
739             compile = false;
740         }
741     }
742     else{
743         // Failed on lexeme_is() expected endif or else
744         ErrorMsg("endif | else");
745         compile = false;
746     }
747 }
748 else{
749     // Failed on Statement()
750     compile = false;
751 }
752 }
753 else{
754     // Failed on lexeme_is("(")
755     ErrorMsg("(");
756     compile = false;
757 }
758 }
759 else{
760     // Failed on Condition()
761     compile = false;
762 }
763 }
764 else{
765     // Failed on lexeme_is("(")
766     ErrorMsg("(");
767     compile = false;
768 }
769 }
770 else{
771     // Failed on lexeme_is("if")
772     compile = false;
773 }
774 }
775 return compile;
776 } // End of If()
777
778 // ----- Return -----
779 // -----
780 bool Return(void)
781 {
782     // Production Rule
783     parserFile << "<Return> -> return ; | return <Expression> ;\n";
784
785     // Create compile flag
786     bool compile = true;
787
788     if(lexeme_is("return")){
789         if(lexeme_is(";")){
790             // <Return> ::= return ;
791         }
792         else if(Expression()){
793             if(lexeme_is(";")){
794                 // <Return> ::= return <Expression> ;
795             }
796             else{
797                 // Failed on lexeme_is(";")
798                 ErrorMsg(";");
799                 compile = false;
800             }
801         }
802     }
803 }
804

```

```

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 } // End of Return()
818
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 } // End of Write()
870
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 } // End of Read()
922
923
924 // ----- While -----
925 // -----
926 bool While(void)
927 {
928
929     // Production Rule
930     parserFile << "<While> -> while (<Condition>) <Statement>\n";
931
932     // Create compile flag
933     bool compile = true;
934
935     // Must start with keyword "while"
936     if(lexeme_is("while")){
937         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     // Production Rule
981     parserFile << "<Expression> -> <Term> <Expression Prime>\n";
982
983     // Create compile flag
984     bool compile = true;
985
986     if(Term()){
987         if(Expression_Prime()){
988             // <Expression> ::= <Term> <Expression Prime>
989         }
990         else{
991             // Failed on Expression_Prime()
992             compile = false;
993         }
994     }
995     else{
996         // Failed on Term()
997         compile = false;
998     }
999 }
1000
1001 return compile;
1002 } // End of Expression()
1003
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 } // End of Condition()
1039
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 } // End of Relop()
1065
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 } // End of Expression_Prime()
1103
1104
1105 // ----- Term -----
1106 // -----
1107 bool Term(void)
1108 {
1109     // Production Rule
1110     parserFile << "<Term> -> <Factor> <Term Prime>\n";
1111
1112     // Create compile flag
1113     bool compile = true;
1114
1115     if(Factor()){
1116         if(Term_Prime()){
1117             // <Term> ::= <Factor> <Term Prime>
1118         }
1119         else{
1120             // Failed on Term_Prime()
1121             compile = false;
1122         }
1123     }
1124     else{
1125         // Failed on Factor()
1126         compile = false;
1127     }
1128
1129     return compile;
1130 } // End of Term()
1131
1132
1133 // ----- Term_Prime -----
1134 // -----
1135 bool Term_Prime(void)
1136 {

```

```

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_funcs.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_funcs.h
```



```

1 Token: separator Lexeme: $$
2 <Rat16F> -> $$ <Opt Function Definitions>
3 $$ <Opt Declaration List> <Statement List> $$
4
5 Token: separator Lexeme: $$
6 <Opt Function Definitions> -> <Function Definitions> | <Empty>
7 <Function Definitions> -> <Function> | <Function> <Function Definitions>
8 <Function> -> function <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>
9
10 Token: separator Lexeme: {
11 <Opt Declaration List> -> <Declaration List> | <Empty>
12 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
13 <Declaration> -> <Qualifier> <IDs>
14 <Qualifier> -> integer | boolean | real
15 <Statement List> -> <Statement> | <Statement> <Statement List>
16 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
17 <Compound> -> { <Statement List> }
18
19 Token: identifier Lexeme: a
20 <Statement List> -> <Statement> | <Statement> <Statement List>
21 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
22 <Compound> -> { <Statement List> }
23 <Assign> -> <Identifier> := <Expression>;
24
25 Token: operator Lexeme: :=
26
27 Token: identifier Lexeme: b
28 <Expression> -> <Term> <Expression Prime>
29 <Term> -> <Factor> <Term Prime>
30 <Factor> -> - <Primary> | <Primary>
31 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false
32
33 Token: operator Lexeme: +
34 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
35 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
36
37 Token: identifier Lexeme: c
38 <Term> -> <Factor> <Term Prime>
39 <Factor> -> - <Primary> | <Primary>
40 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true | false
41
42 Token: separator Lexeme: ;
43 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
44 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
45
46 Token: separator Lexeme: }
47 <If> -> if (<Condition>) <Statement> endif |
48 if (<Condition>) <Statement> else <Statement> endif
49 <Return> -> return ; | return <Expression> ;
50 <Write> -> print (<Expression>);
51 <Read> -> read (<IDs>);
52 <While> -> while (<Condition>) <Statement>
53 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
54 <Compound> -> { <Statement List> }
55 <Assign> -> <Identifier> := <Expression>;
56 <If> -> if (<Condition>) <Statement> endif |
57 if (<Condition>) <Statement> else <Statement> endif
58 <Return> -> return ; | return <Expression> ;
59 <Write> -> print (<Expression>);
60 <Read> -> read (<IDs>);
61 <While> -> while (<Condition>) <Statement>
62
63 Token: separator Lexeme: $$
64 <Assign> -> <Identifier> := <Expression>;
65 <If> -> if (<Condition>) <Statement> endif |
66 if (<Condition>) <Statement> else <Statement> endif
67 <Return> -> return ; | return <Expression> ;

```

```
68 <Write> -> print (<Expression>);
69 <Read> -> read (<IDs>);
70 <While> -> while (<Condition>) <Statement>
71 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
72 <Compound> -> { <Statement List> }
73 <Assign> -> <Identifier> := <Expression>;
74 <If> -> if (<Condition>) <Statement> endif |
75 if (<Condition>) <Statement> else <Statement> endif
76 <Return> -> return ; | return <Expression> ;
77 <Write> -> print (<Expression>);
78 <Read> -> read (<IDs>);
79 <While> -> while (<Condition>) <Statement>
80
81 Token: Lexeme:
```

```
1  $$
2  $$
3  {
4  a := b + c;
5  }
6  $$
```

```

1  Token: separator Lexeme: $$
2  <Rat16F> -> $$ <Opt Function Definitions>
3  $$ <Opt Declaration List> <Statement List> $$
4
5  Token: keyword Lexeme: function
6  <Opt Function Definitions> -> <Function Definitions> | <Empty>
7  <Function Definitions> -> <Function> | <Function> <Function Definitions>
8  <Function> -> function <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>
9
10 Token: identifier Lexeme: Subtract
11
12 Token: separator Lexeme: [
13
14 Token: identifier Lexeme: imVal
15 <Opt Parameter List> -> <Parameter List> | <Empty>
16 <Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
17 <Parameter> -> <IDs> : <Qualifier>
18 <IDs> -> <Identifier> | <Identifier>, <IDs>
19
20 Token: separator Lexeme: :
21
22 Token: keyword Lexeme: integer
23 <Qualifier> -> integer | boolean | real
24
25 Token: separator Lexeme: ]
26
27 Token: keyword Lexeme: real
28 <Opt Declaration List> -> <Declaration List> | <Empty>
29 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
30 <Declaration> -> <Qualifier> <IDs>
31 <Qualifier> -> integer | boolean | real
32
33 Token: identifier Lexeme: retVal92
34 <IDs> -> <Identifier> | <Identifier>, <IDs>
35
36 Token: separator Lexeme: ;
37
38 Token: separator Lexeme: {
39 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
40 <Declaration> -> <Qualifier> <IDs>
41 <Qualifier> -> integer | boolean | real
42 <Body> -> { <Statement List> }
43
44 Token: identifier Lexeme: imVal
45 <Statement List> -> <Statement> | <Statement> <Statement List>
46 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
47 <Compound> -> { <Statement List> }
48 <Assign> -> <Identifier> := <Expression>;
49
50 Token: operator Lexeme: :=
51
52 Token: identifier Lexeme: imVal
53 <Expression> -> <Term> <Expression Prime>
54 <Term> -> <Factor> <Term Prime>
55 <Factor> -> - <Primary> | <Primary>
56 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
57
58 Token: operator Lexeme: -
59 <Term Prime> -> * <Factor> <Term Prime> | / <Factor> <Term Prime> | epsilon
60 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
61
62 Token: separator Lexeme: (
63 <Term> -> <Factor> <Term Prime>
64 <Factor> -> - <Primary> | <Primary>
65 <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>);

```

```

129 <While> -> while (<Condition>) <Statement>
130 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
131 <Compound> -> { <Statement List> }
132 <Assign> -> <Identifier> := <Expression>;
133 <If> -> if (<Condition>) <Statement> endif |
134 if (<Condition>) <Statement> else <Statement> endif
135 <Return> -> return ; | return <Expression> ;
136 <Write> -> print (<Expression>);
137 <Read> -> read (<IDs>);
138 <While> -> while (<Condition>) <Statement>
139
140 Token: separator Lexeme: $$
141 <Function Definitions> -> <Function> | <Function> <Function Definitions>
142 <Function> -> function <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>
143
144 Token: keyword Lexeme: integer
145 <Opt Declaration List> -> <Declaration List> | <Empty>
146 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
147 <Declaration> -> <Qualifier> <IDs>
148 <Qualifier> -> integer | boolean | real
149
150 Token: identifier Lexeme: low_av
151 <IDs> -> <Identifier> | <Identifier>, <IDs>
152
153 Token: separator Lexeme: ,
154
155 Token: identifier Lexeme: high_av
156
157 Token: separator Lexeme: ;
158
159 Token: keyword Lexeme: read
160 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
161 <Declaration> -> <Qualifier> <IDs>
162 <Qualifier> -> integer | boolean | real
163 <Statement List> -> <Statement> | <Statement> <Statement List>
164 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
165 <Compound> -> { <Statement List> }
166 <Assign> -> <Identifier> := <Expression>;
167 <If> -> if (<Condition>) <Statement> endif |
168 if (<Condition>) <Statement> else <Statement> endif
169 <Return> -> return ; | return <Expression> ;
170 <Write> -> print (<Expression>);
171 <Read> -> read (<IDs>);
172
173 Token: separator Lexeme: (
174
175 Token: identifier Lexeme: low_av
176 <IDs> -> <Identifier> | <Identifier>, <IDs>
177
178 Token: separator Lexeme: ,
179
180 Token: identifier Lexeme: high_av
181
182 Token: separator Lexeme: )
183
184 Token: separator Lexeme: ;
185
186 Token: keyword Lexeme: while
187 <While> -> while (<Condition>) <Statement>
188
189 Token: separator Lexeme: (
190
191 Token: identifier Lexeme: low_av
192 <Condition> -> <Expression> <Relop> <Expression>
193 <Expression> -> <Term> <Expression Prime>
194 <Term> -> <Factor> <Term Prime>
195 <Factor> -> - <Primary> | <Primary>

```

```

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

```

```
260 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
261
262 Token: separator Lexeme: [
263
264 Token: identifier Lexeme: low_av
265 <IDs> -> <Identifier> | <Identifier>, <IDs>
266
267 Token: separator Lexeme: ]
268
269 Token: separator Lexeme: )
270 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
271 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
272
273 Token: separator Lexeme: ;
274
275 Token: separator Lexeme: }
276 <Read> -> read (<IDs>);
277 <While> -> while (<Condition>) <Statement>
278 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
279 <Compound> -> { <Statement List> }
280 <Assign> -> <Identifier> := <Expression>;
281 <If> -> if (<Condition>) <Statement> endif |
282 if (<Condition>) <Statement> else <Statement> endif
283 <Return> -> return ; | return <Expression> ;
284 <Write> -> print (<Expression>);
285 <Read> -> read (<IDs>);
286 <While> -> while (<Condition>) <Statement>
287
288 Token: separator Lexeme: $$
289 <Assign> -> <Identifier> := <Expression>;
290 <If> -> if (<Condition>) <Statement> endif |
291 if (<Condition>) <Statement> else <Statement> endif
292 <Return> -> return ; | return <Expression> ;
293 <Write> -> print (<Expression>);
294 <Read> -> read (<IDs>);
295 <While> -> while (<Condition>) <Statement>
296 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
297 <Compound> -> { <Statement List> }
298 <Assign> -> <Identifier> := <Expression>;
299 <If> -> if (<Condition>) <Statement> endif |
300 if (<Condition>) <Statement> else <Statement> endif
301 <Return> -> return ; | return <Expression> ;
302 <Write> -> print (<Expression>);
303 <Read> -> read (<IDs>);
304 <While> -> while (<Condition>) <Statement>
305
306 Token: Lexeme:
```



```
1  $$
2  function Subtract[imVal:integer]
3    real retVal92;
4    {
5      imVal := imVal - (2 * imVal);
6      retVal92:=imVal;
7      return retVal92;
8    }
9  $$
10     integer  low_av,high_av;
11     read(low_av,high_av);
12     while (low_av <= high_av)
13     {
14       print(low_av);
15       print(Subtract[low_av]);
16     }
17  $$
```

```
1 Token: separator Lexeme: $$
2 <Rat16F> -> $$ <Opt Function Definitions>
3 $$ <Opt Declaration List> <Statement List> $$
4
5 Token: keyword Lexeme: function
6 <Opt Function Definitions> -> <Function Definitions> | <Empty>
7 <Function Definitions> -> <Function> | <Function> <Function Definitions>
8 <Function> -> function <Identifier> [ <Opt Parameter List> ] <Opt Declaration List> <Body>
9
10 Token: identifier Lexeme: Hello
11
12 Token: separator Lexeme: [
13
14 Token: identifier Lexeme: me
15 <Opt Parameter List> -> <Parameter List> | <Empty>
16 <Parameter List> -> <Parameter> | <Parameter> , <Parameter List>
17 <Parameter> -> <IDs> : <Qualifier>
18 <IDs> -> <Identifier> | <Identifier>, <IDs>
19
20 Token: separator Lexeme: :
21
22 Token: keyword Lexeme: real
23 <Qualifier> -> integer | boolean | real
24
25 Token: separator Lexeme: ,
26
27 Token: identifier Lexeme: alice
28 <Parameter> -> <IDs> : <Qualifier>
29 <IDs> -> <Identifier> | <Identifier>, <IDs>
30
31 Token: separator Lexeme: :
32
33 Token: keyword Lexeme: integer
34 <Qualifier> -> integer | boolean | real
35
36 Token: separator Lexeme: ,
37
38 Token: identifier Lexeme: bob
39 <Parameter> -> <IDs> : <Qualifier>
40 <IDs> -> <Identifier> | <Identifier>, <IDs>
41
42 Token: separator Lexeme: :
43
44 Token: keyword Lexeme: boolean
45 <Qualifier> -> integer | boolean | real
46
47 Token: separator Lexeme: ]
48
49 Token: keyword Lexeme: boolean
50 <Opt Declaration List> -> <Declaration List> | <Empty>
51 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
52 <Declaration> -> <Qualifier> <IDs>
53 <Qualifier> -> integer | boolean | real
54
55 Token: identifier Lexeme: modest
56 <IDs> -> <Identifier> | <Identifier>, <IDs>
57
58 Token: separator Lexeme: ;
59
60 Token: keyword Lexeme: boolean
61 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
62 <Declaration> -> <Qualifier> <IDs>
63 <Qualifier> -> integer | boolean | real
64
65 Token: identifier Lexeme: mouse
66 <IDs> -> <Identifier> | <Identifier>, <IDs>
67
```

```
68 Token: separator Lexeme: ;
69
70 Token: keyword Lexeme: real
71 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
72 <Declaration> -> <Qualifier> <IDs>
73 <Qualifier> -> integer | boolean | real
74
75 Token: identifier Lexeme: pink
76 <IDs> -> <Identifier> | <Identifier>, <IDs>
77
78 Token: separator Lexeme: ;
79
80 Token: keyword Lexeme: integer
81 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
82 <Declaration> -> <Qualifier> <IDs>
83 <Qualifier> -> integer | boolean | real
84
85 Token: identifier Lexeme: floyd
86 <IDs> -> <Identifier> | <Identifier>, <IDs>
87
88 Token: separator Lexeme: ;
89
90 Token: separator Lexeme: {
91 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
92 <Declaration> -> <Qualifier> <IDs>
93 <Qualifier> -> integer | boolean | real
94 <Body> -> { <Statement List> }
95
96 Token: identifier Lexeme: alice
97 <Statement List> -> <Statement> | <Statement> <Statement List>
98 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
99 <Compound> -> { <Statement List> }
100 <Assign> -> <Identifier> := <Expression>;
101
102 Token: operator Lexeme: :=
103
104 Token: identifier Lexeme: me
105 <Expression> -> <Term> <Expression Prime>
106 <Term> -> <Factor> <Term Prime>
107 <Factor> -> - <Primary> | <Primary>
108 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
109
110 Token: separator Lexeme: ;
111 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
112 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
113
114 Token: keyword Lexeme: if
115 <If> -> if (<Condition>) <Statement> endif |
116 if (<Condition>) <Statement> else <Statement> endif
117
118 Token: separator Lexeme: (
119
120 Token: identifier Lexeme: pink
121 <Condition> -> <Expression> <Relop> <Expression>
122 <Expression> -> <Term> <Expression Prime>
123 <Term> -> <Factor> <Term Prime>
124 <Factor> -> - <Primary> | <Primary>
125 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
126
127 Token: operator Lexeme: +
128 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
129 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
130
131 Token: identifier Lexeme: floyd
132 <Term> -> <Factor> <Term Prime>
```

```

133 <Factor> -> - <Primary> | <Primary>
134 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
135
136 Token: operator Lexeme: >
137 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
138 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
139 <Relop> -> = | /= | > | < | => | <=
140
141 Token: identifier Lexeme: modest
142 <Expression> -> <Term> <Expression Prime>
143 <Term> -> <Factor> <Term Prime>
144 <Factor> -> - <Primary> | <Primary>
145 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
146
147 Token: operator Lexeme: /
148 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
149
150 Token: identifier Lexeme: mouse
151 <Factor> -> - <Primary> | <Primary>
152 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
153
154 Token: separator Lexeme: )
155 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
156 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
157
158 Token: identifier Lexeme: me
159 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
160 <Compound> -> { <Statement List> }
161 <Assign> -> <Identifier> := <Expression>;
162
163 Token: operator Lexeme: :=
164
165 Token: identifier Lexeme: alice
166 <Expression> -> <Term> <Expression Prime>
167 <Term> -> <Factor> <Term Prime>
168 <Factor> -> - <Primary> | <Primary>
169 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
    false
170
171 Token: separator Lexeme: ;
172 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
173 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
174
175 Token: keyword Lexeme: else
176 <If> -> if (<Condition>) <Statement> endif |
177 if (<Condition>) <Statement> else <Statement> endif
178 <Return> -> return ; | return <Expression> ;
179 <Write> -> print (<Expression>);
180 <Read> -> read (<IDs>);
181 <While> -> while (<Condition>) <Statement>
182
183 Token: identifier Lexeme: me
184 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
185 <Compound> -> { <Statement List> }
186 <Assign> -> <Identifier> := <Expression>;
187
188 Token: operator Lexeme: :=
189
190 Token: identifier Lexeme: modest
191 <Expression> -> <Term> <Expression Prime>
192 <Term> -> <Factor> <Term Prime>
193 <Factor> -> - <Primary> | <Primary>
194 <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

```

```

260 <Opt Declaration List> -> <Declaration List> | <Empty>
261 <Declaration List> -> <Declaration> ; | <Declaration> ; <Declaration List>
262 <Declaration> -> <Qualifier> <IDs>
263 <Qualifier> -> integer | boolean | real
264 <Statement List> -> <Statement> | <Statement> <Statement List>
265 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
266 <Compound> -> { <Statement List> }
267 <Assign> -> <Identifier> := <Expression>;
268 <If> -> if (<Condition>) <Statement> endif |
269 if (<Condition>) <Statement> else <Statement> endif
270 <Return> -> return ; | return <Expression> ;
271 <Write> -> print (<Expression>);
272 <Read> -> read (<IDs>);
273 <While> -> while (<Condition>) <Statement>
274
275 Token: separator Lexeme: (
276
277 Token: integer Lexeme: 1
278 <Condition> -> <Expression> <Relop> <Expression>
279 <Expression> -> <Term> <Expression Prime>
280 <Term> -> <Factor> <Term Prime>
281 <Factor> -> - <Primary> | <Primary>
282 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
283
284 Token: operator Lexeme: >
285 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
286 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
287 <Relop> -> = | /= | > | < | => | <=
288
289 Token: integer Lexeme: 0
290 <Expression> -> <Term> <Expression Prime>
291 <Term> -> <Factor> <Term Prime>
292 <Factor> -> - <Primary> | <Primary>
293 <Primary> -> <Identifier> | <Integer> | <Identifier> [<IDs>] | (<Expression>) | <Real> | true |
false
294
295 Token: separator Lexeme: )
296 <Term Prime> -> * <Factor> <Term Prime> | / Factor <Term Prime> | epsilon
297 <Expression Prime> -> +<Term> <Expression Prime> | -<Term> <Expression Prime> | epsilon
298
299 Token: separator Lexeme: {
300 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
301 <Compound> -> { <Statement List> }
302
303 Token: keyword Lexeme: read
304 <Statement List> -> <Statement> | <Statement> <Statement List>
305 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
306 <Compound> -> { <Statement List> }
307 <Assign> -> <Identifier> := <Expression>;
308 <If> -> if (<Condition>) <Statement> endif |
309 if (<Condition>) <Statement> else <Statement> endif
310 <Return> -> return ; | return <Expression> ;
311 <Write> -> print (<Expression>);
312 <Read> -> read (<IDs>);
313
314 Token: separator Lexeme: (
315
316 Token: identifier Lexeme: Hello
317 <IDs> -> <Identifier> | <Identifier>, <IDs>
318
319 Token: separator Lexeme: )
320
321 Token: separator Lexeme: ;
322
323 Token: separator Lexeme: }
324 <While> -> while (<Condition>) <Statement>

```

```
325 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
326 <Compound> -> { <Statement List> }
327 <Assign> -> <Identifier> := <Expression>;
328 <If> -> if (<Condition>) <Statement> endif |
329 if (<Condition>) <Statement> else <Statement> endif
330 <Return> -> return ; | return <Expression> ;
331 <Write> -> print (<Expression>);
332 <Read> -> read (<IDs>);
333 <While> -> while (<Condition>) <Statement>
334
335 Token: separator Lexeme: $$
336 <Assign> -> <Identifier> := <Expression>;
337 <If> -> if (<Condition>) <Statement> endif |
338 if (<Condition>) <Statement> else <Statement> endif
339 <Return> -> return ; | return <Expression> ;
340 <Write> -> print (<Expression>);
341 <Read> -> read (<IDs>);
342 <While> -> while (<Condition>) <Statement>
343 <Statement> -> <Compound> | <Assign> | <If> | <Return> | <Write> | <Read> | <While>
344 <Compound> -> { <Statement List> }
345 <Assign> -> <Identifier> := <Expression>;
346 <If> -> if (<Condition>) <Statement> endif |
347 if (<Condition>) <Statement> else <Statement> endif
348 <Return> -> return ; | return <Expression> ;
349 <Write> -> print (<Expression>);
350 <Read> -> read (<IDs>);
351 <While> -> while (<Condition>) <Statement>
352
353 Token: Lexeme:
```

```
1  $$
2  function Hello[me:real, alice:integer, bob:boolean]
3
4  boolean modest; boolean mouse;
5  real pink;
6  integer floyd;
7  {
8      alice := me;
9      if(pink + floyd > modest/mouse)
10         me := alice;
11     else
12         me := modest + mouse;
13     endif
14
15     return me[modest, alice];
16 }
17 $$
18 while(1 > 0)
19 {
20     read(Hello);
21 }
22 $$
```