```
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:
```