CS315
PROJECT 2 REPORT
TEAM 16

ERHAN ER - 21801809 - SECTION 3
MUHAMMET ABDULLAH KOÇ - 21802832 - SECTION 3
MURAT FURKAN UĞURLU - 21802062 - SECTION 3

LANGUAGE NAME: GRONDSTOF

1. BNF Description of Grondstof

1.1 Program

```
<program> ::= GRD-START {<stmt_list>}
<stmt_list> ::= <stmt> | <stmt_list><stmt>
<stmt> ::= <expression> | <comment>
<expression> ::= <conditional> | <non_conditional>

<non_conditional> ::= <declaration_stmt><end_expression>
                    | <assignment_stmt><end_expression>
                    | <input_stmt><end_expression>
                    | <output_stmt><end_expression>
                    | <loop_stmt>
                    | <func_declaration>
                    | <func_call><end_expression>
                    | <return_stmt><end_expression>
                    | <primitive_funcs><end_expression>
```

1.2 If Statement

```
<conditional> ::= if ( <logic_expr> ) { <stmt_list> }
             | if ( <logic_expr> ) { <stmt_list> } else { <stmt_list> }
             | if ( <logic_expr> ) { <stmt_list> } <else_if>

<else_if> ::= else_if ( <logic_expr> ) { <stmt_list> }
       | else_if ( <logic_expr> ) { <stmt_list> } else { <stmt_list> }
       | else_if ( <logic_expr> ) { <stmt_list> } <else_if>
```

1.3 Non-If Statement

Declaration Statement

```
<declaration_stmt> ::= <initialized_declaration>
                         | <not_initialized_decleration>
<initialized_decleration> ::= <int_declaration>
                              | <double_declaration>
                              | <char_declaration>
                              | <string_declaration>
                              | <boolean_declaration>

<int_declaration> ::= int <identifier> = <func_call>
                    | int <identifier> = <arithmetic_exprs>

<double_declaration> ::= double <identifier> = <func_call>
```

```
                  | double <identifier> = <arithmetic_exprs>

<char_decleration> ::= char <identifier> = <func_call>
                  | char <identifier> = <arithmetic_exprs>

<string_declaration> ::= string <identifier> = <func_call>
                  | string <identifier> = <arithmetic_exprs>

<boolean_declaration> ::= boolean <identifier> = <func_call>
                  | boolean <identifier> = <arithmetic_exprs>


<not_initialized_declaration> ::= int <identifier>
                                | double <identifier>
                                | char <identifier>
                                | string <identifier>
                                | boolean <identifier>
```

Assignment

```
<assignment_stmt> ::= <identifier> = <func_call>
                  | <identifier> = <logic_expr>
                  | <identifier> = <arithmetic_exprs>
                  | <increment_expr>
                  | <decrement_expr>
```

Function Declaration

```
<func_declaration> ::=
              int function <identifier> (<param_list>) {<func_body>}
            | double function <identifier> (<param_list>) {<func_body>}
            | boolean function <identifier> (<param_list>) {<func_body>}
            | string function <identifier> (<param_list>) {<func_body>}
            | char function <identifier> (<param_list>) {<func_body>}
            | void function <identifier> (<param_list>) {<func_body>}

<param_list> ::= <empty> | <param> | <param_list>, <param>
<param> ::= int <identifier> | string <identifier>
            | double <identifier> | char <identifier>
            | boolean <identifier> | <const>
<func_body> ::= <stmt_list> <return_stmt> | <stmt_list>
```

Function Call

```
<func_call> ::= <identifier> (<identifier_list>)
            | <identifier> (<logic_expression_list)
<logic_expression_list> ::= <logic_expr>
                        | <logic_expression_list>, <logic_expr>
<identifier_list> ::= <empty> | <arithmetic_exprs> | <identifier_list>,
                  <arithmetic_exprs>
```

Loop Statement

```
<loop_stmt> ::= <while_stmt> | <for_stmt>
<while_stmt> ::= while (<logic_expr>) {<stmt_list>}
<for_stmt> ::= for ( <initialized_decleration>; <logic_expr>;
            <arithmetic_exprs> ) {<stmt_list>}
            | for ( <initialized_decleration>; <logic_expr>;
            <increment_expr> ) {<stmt_list>}
            | for ( <initialized_decleration>; <logic_expr>;
            <decrement_expr> ) {<stmt_list>}
```

Return Statement

```
<return_stmt> ::= return <func_call>
                  | return <const>
                  | return <identifier>
                  | return
```

Input Output Statement

```
<input_stmt> ::= input()
<output_stmt> ::= out(<identifier>) | outln(<identifier>) | out(<const>)
            | outln(<const>) | out(<func_call>) | outln(<func_call>)
            | out(<primitive_funcs>) | outln(<primitive_funcs>)
```

Comment Statement

```
<comment> ::= \/\*\*.\*\*\/
```

Primitive Functions

```
<primitive_funcs> ::= <read_altitude> | <read_heading>
                  | <read_temperature> | <turn_heading_one_degree>
                  | <climb_ver> | <move_hor_one_ms>
```

```
                  | <connect_to_computer> | <disconnect_to_computer>
                  | <turn_on_off_spray> | <set_altitude> | <set_heading>
                  | <get_battery_percentage> | <set_height>
                  | <wait_next_instruction>
<read_altitude> ::= getAltitude()
<read_heading> ::= getHeading()
<read_temperature> ::= getTemperature()
<turn_heading_one_degree> ::= turnLeftRight(<logic_expr>)
                              | turnLeftRight(<identifier>)
<climb_ver> ::= climbUpStopDown(<logic_expr>, <logic_expr>)
            | climbUpStopDown(<identifier>, <logic_expr>)
            | climbUpStopDown(<logic_expr>, <identifier>)
            | climbUpStopDown(<identifier>, <identifier>)
<move_hor_one_ms> ::= moveForwardStopBackward(<logic_expr>, <logic_expr>)
            | moveForwardStopBackward(<identifier>, <logic_expr>)
            | moveForwardStopBackward(<logic_expr>, <identifier>)
            | moveForwardStopBackward(<identifier>, <identifier>)
<connect_to_computer> ::= connect()
<disconnect_to_computer> ::= disconnect()
<turn_on_off_spray> ::= changeSpray(<logic_expr>)
                  | changeSpray(<identifer>)
<set_altitude> ::= setAltitude(<arithmetic_exprs>)
<set_heading> ::= setHeading(<arithmetic_exprs>)
<get_battery_percentage> ::= getBatteryPerc()
<set_height> ::= setHeight()
<wait_next_instruction> ::= waitNextInst( <arithmetic_exprs> )
```

1.4 Expressions

```
<logic_expr> ::= <and_out>

<and_out> ::= <or_out> | <and_out> && <or_out>

<or_out> ::= <and_or_in> | <and_or_in> || <or_out>

<and_or_in> ::= <relational_expr>

<relational_expr> ::= <gt_op>
                  | <lt_op>
                  | <gte_op>
                  | <lte_op>
                  | <eq_op>
                  | <not_eq_op>
                  | <boolean>

<gt_op> ::= <arithmetic_exprs> > <arithmetic_exprs>
```

```
<lt_op> ::= <arithmetic_exprs> < <arithmetic_exprs>

<gte_op> ::= <arithmetic_exprs> >= <arithmetic_exprs>

<lte_op> ::= <arithmetic_exprs> <= <arithmetic_exprs>

<eq_op> ::= <arithmetic_exprs> == <arithmetic_exprs>

<not_eq_op> ::= <arithmetic_exprs> != <arithmetic_exprs>



<arithmetic_exprs> ::= <general_expr> | ( <general_expr> )

<general_expr> ::= <prior_expr> | <general_expr> + <prior_expr>
                  | <general_expr> - <prior_expr>
<prior_expr> ::= <term> | <prior_expr> * <term> | <prior_expr> / <term>
                  | <prior_expr> % <term>
<term> ::= <input_stmt> | <int> | <double> | <string> | <char>
      | <identifier> | (<func_call>) | <primitive_funcs>

<increment_expr> ::= <identifier>++ | ++<identifier>

<decrement_expr> ::= <identifier>-- | --<identifier>
```

1.5 Regular Expressions

```
<sign> ::= [+-]
<digit> ::= [0-9]
<alphabetic> ::= [a-zA-Z_]
<dot> ::= \.
<alphanumeric> ::= <digit> | <alphabetic> | <alphanumeric><digit>
                  | <alphanumeric><alphabetic>
<identifier> ::= <alphabetic> | <alphabetic><alphanumeric>
<string> ::= \”[^\”]*\”
<int> ::= <sign><digit> | <digit>
<double> ::= <sign><digit><dot><digit> | <digit><dot><digit> |
           <dot><digit>
<boolean> ::= true | false
<char> ::= ’[^\’]’
<const> ::= <string> | <int> | <double> | <boolean> | <char>
<empty> ::=
<end_expression> ::= \;
```

2. Detailed Explanation of the Language

1. **<program> ::= GRD-START {<stmt_list>}**

   Program is defined with a set of statements (<stmt_list>). Our program has a reserved keyword for starting. Statements will be inside of curly braces.

2. **<stmt_list> ::= <stmt> | <stmt_list><stmt>**

   A statement list can either be one statement or consist of one or more statements which are defined in the statement set (<stmt>).

3. **<empty> ::=**

   It is a regular expression which is assigned to nothing so that declarations do not have to be assigned to a value.

4. **<stmt> ::= <expression> | <comment>**

   This stands for a statement and a statement can either be an expression or comment.

5. **<expression> ::= <conditional> | <non_conditional>**

   An expression can either be a conditional or non-conditional statement.

6. **<end_expression> ::= \;**

   This statement is used to indicate the end of an expression.

7. **<comment> ::= \/**.**\/**

Comment is defined with “/** **/” e.g., /**Example comment**/

8. **<conditional> ::= if (<logic_expr>) { <stmt_list> }**
   | if (<logic_expr>) { <stmt_list> } else { <stmt_list> }
   | if (<logic_expr>) { <stmt_list> } <else_if>

   A conditional stands for if statements in our program. It can be just a single if statement, if-else statement or it can be if-if else statement.

9. **<else_if> ::= else_if (<logic_expr>) { <stmt_list> }**
   | else_if (<logic_expr>) { <stmt_list> } else { <stmt_list> }
   | else_if (<logic_expr>) { <stmt_list> } <else_if>

   This statement is created for the purpose of having an “else if” statement. This function provides our program to have “else if” statements such that the program can have consecutive “else if” statements, and also end up with an “else” statement after “else if” statements.

10. **<non_conditional> ::= <declaration_stmt><end_expression>**
    | <assignment_stmt><end_expression>
    | <input_stmt><end_expression>
    | <output_stmt><end_expression>
    | <loop_stmt>
    | <func_declaration>
    | <func_call><end_expression>
    | <return_stmt><end_expression>
    | <primitive_funcs><end_expression>

    A non-conditional stands for unconditional statements (i.e. statements which are not an if statement). It can be a declaration, assignment , input-output, loop, function, pre-defined functions or return statement.

11. **<declaration_stmt> ::= <initialized_declaration>**
    | <not_initialized_decleration>

    A declaration statement can be an initialization statement (e.g., int i = 7;) or an assignment statement (e.g., i = 24;).

12. **<assignment_stmt> ::= <identifier> = <func_call>**
    | <identifier> = <logic_expr>
    | <identifier> = <arithmetic_exprs>
    | <increment_expr>
    | <decrement_expr>

Assignment statement is used for assigning a new value to an already created variable (e.g., i = 48;)

13. **<input_stmt> ::= input()**

This statement is used for getting input from the user.

14. **<output_stmt> ::= out(<identifier>) | outln(<identifier>) | out(<const>) | outln(<const>) | out(<func_call>) | outln(<func_call>) | out(<primitive_funcs>) | outln(<primitive_funcs>)**

This statement is used to print integers, strings, chars, booleans and doubles. Out function prints the parameter but does not create a new line. However, outln function creates a new line after printing the parameter.

15. **<loop_stmt> ::= <while_stmt> | <for_stmt>**

Loop statement can be either a “for loop” or a “while loop”. We used while and for loop to increase readability and writability.

16. **<func_declaration> ::=**

int function <identifier> (<param_list>) {<func_body>}
| double function <identifier> (<param_list) {<func_body>}
| boolean function <identifier> (<param_list>) {<func_body>}
| string function <identifier> (<param_list>) {<func_body>}
| char function <identifier> (<param_list>) {<func_body>}
| void function <identifier> (<param_list>) {<func_body>}

Function declaration is created for the purpose of providing users with creating their own functions. Users can enter more than one parameter defined in “<param_list>”.

17. **<func_call> ::= <identifier> (<identifier_list>)**

| <identifer> (<logic_expression_list>)

This statement is used for calling functions with their names and parameters.

18. **<return_stmt> ::= return <func_call>**

| return <const>
| return <identifier>
| return

This statement was created for the purpose of providing users with the ability of returning a value in their functions. A return type can be a function call (e.g., return getAltitude();), a const, an identifier. Also, it may not return anything.

19. **<logic_expr> ::= <and_out>**

A logic expression can be an “<and_out>” expression.

20. **<initialized_declaration> ::= <int_declaration>**
| <double_declaration>
| <char_declaration>
| <string_declaration>
| <boolean_declaration>

Initialized declaration statement is created for the purpose of creating and initializing variables which are int, double, char, string and boolean.

21. **<not_initialized_decleration> ::= int <identifier>**
| double <identifier>
| char <identifier>
| string <identifier>
| boolean <identifier>

Non-initialized declaration statement is created for the purpose of creating variables which are int, double, char, string and boolean without an initialization.

22. **<int_declaration> ::= int <identifier> = <func_call>**
| int <identifier> = <arithmetic_exprs>

This statement is created for the purpose of creating and initializing integer variables. An integer variable can be assigned to an integer number, another integer variable, a function return or an input statement.

23. **<double_declaration> ::= double <identifier> = <func_call>**
| double <identifier> = <arithmetic_exprs>

This statement is created for the purpose of creating and initializing double variables. A double variable can be assigned to a double number, another double variable, a function return or an input statement.

24. **<char_declaration> ::= char <identifier> = <func_call>**
| char <identifier> = <arithmetic_exprs>

This statement is created for the purpose of creating and initializing char variables. A char variable can be assigned to a character , another char variable, a function return or an input statement.

25. **<string_declaration> ::= string <identifier> = <func_call>**
| string <identifier> = <arithmetic_exprs>

This statement is created for the purpose of creating and initializing string variables. A string variable can be assigned to a string, another string variable, a function return or an input statement.

26. **<boolean_declaration> ::= boolean <identifier> = <func_call>**
| boolean <identifier> = <arithmetic_exprs>

This statement is created for the purpose of creating and initializing boolean variables. A boolean variable can be assigned to a boolean statement, another boolean variable, a function return or an input statement.

27. **<param_list> ::= <empty> | <param> | <param_list>, <param>**

This set is defined for the purpose of adding parameters to the functions created by the users. Inside the function, a user can define parameters from this statement (<param_list>).

28. **<func_body> ::= <stmt_list> <return_stmt>**
| <stmt_list>

This set is defined for the purpose of filling the functions' body. A function body can be filled with either function statements (<func_statements>) with a return statement (<return_stmt>) to return a value or function call or without a return statement.

29. **<param> ::= int <identifier> | string <identifier>**
| double <identifier> | char <identifier>
| boolean <identifier> | <const>

This set is defined to indicate what can be a parameter to be used in functions. A parameter can either be an identifier or be a const (string, int, boolean or char)

30. **<logic_expression_list> ::= <logic_expression>**
| <logic_expression_list>, <logic_expression>

This statement is used for representing logic expression and a logic expression list.

31. **<identifier_list> ::= <empty> | <identifier>**
| <identifier_list>,<identifier>

This statement is used for representing empty, an identifier and an identifier list.

32. **<while_stmt> ::= while (<logic_expr>) {<stmt_list>}**

This statement was created for the purpose of declaring while loops. A while loop takes a logic expression <logic_expr> to set the condition, and a body. The body consists of one or more elements from the statement list <stmt_list>.

33. **<for_stmt> ::= for (<initialized_decleration>; <logic_expr>;**
<arithmetic_exprs>){<stmt_list>}
| for (<initialized_decleration>; <logic_expr>;
<increment_expr>){<stmt_list>}
| for (<initialized_decleration>; <logic_expr>;
<decrement_expr>){<stmt_list>}

This statement was created for the purpose of declaring for loops. A for loop takes <initialized_declaration> to initialize a variable, a logic expression <logic_expr> to have a condition, an arithmetic expression <arithmetic_exprs> to do arithmetic operations, and a body to do operations needed. The body consists of one or more elements from the statement list <stmt_list>.

34. **<arithmetic_exprs> ::= <general_expr> | (<general_expr>)**

This statement is used for representing arithmetic operations such as addition, substracition, multiplication, division, modular, incrementing and decrementing.

35. **<const> ::= <string> | <int> | <double> | <boolean> | <char>**

This statement is defined for primitive types i.e. string, int, double, boolean and char.

36. **<primitive_funcs> ::= <read_altitude> | <read_heading>**
| <read_temperature> | <turn_heading_one_degree>
| <climb_ver> | <move_hor_one_ms>
| <connect_to_computer>
| <disconnect_to_computer>
| <turn_on_off_spray> | <set_altitude>
| <set_heading> | <get_battery_percentage>
| <set_height> | <wait_next_instruction>

This keyword is defined for primitive functions.

37. **<read_altitude> ::= getAltitude()**

This statement is created to indicate the “getAltitude()” function. This function returns the altitude of the point where the drone is currently positioned.

38. **<read_heading> ::= getHeading()**

This statement is created to indicate the “getHeading()” function. This function returns the angle of the direction where the drone is currently headed.

39. **<read_temperature> ::= getTemperature()**

This statement is created to indicate the “getTemperature()” function. This function returns the temperature of the point where the drone is currently positioned.

40. **<turn_heading_one_degree> ::= turnLeftRight(<logic_expr>)**
| turnLeftRight(<identifier>)

This statement is created to indicate the “turnLeftRight(<boolean>)” function. This function makes the drone turn one degree to the left or right. If the boolean value in the parameter is true, the drone turns to the left direction in one degree. Otherwise, the drone turns to the right direction in one degree.

41. **<climb_ver> ::= climbUpStopDown(<logic_expr>, <logic_expr>)**
| climbUpStopDown(<identifier>, <logic_expr>)
| climbUpStopDown(<logic_expr>, <identifier>)
| climbUpStopDown(<identifier>, <identifier>)

This statement is created to indicate the “climbUpStopDown(<boolean>, <boolean>)” function. This function makes the drone climb up, stop or climb down with a speed of 0.1m/s. If the first boolean value in the parameter is true, the drone stops and does not move. Otherwise, the drone climbs up or down according to the second boolean parameter. If the second boolean parameter is true, then the drone climbs up (assuming that the first boolean value is false). If not, the drone climbs down.

42. **<move_hor_one_ms> ::=**
moveForwardStopBackward(<logic_expr>, <logic_expr>)
| moveForwardStopBackward(<identifier>, <logic_expr>)
| moveForwardStopBackward(<logic_expr>, <identifier>)
| moveForwardStopBackward(<identifier>, <identifier>)

This function allows the users to move the drone to forward or backward. The function takes two boolean values. The first one decides whether the drone will stop or move. If it is true, the drone will stop. The second boolean value decides whether the drone will go forward or backward. If it is true, the drone will go forward with a speed of 1 m/s. If it is false, the drone will go backwards with a speed of 1 m/s.

43. **<connect_to_computer> ::= connect()**

This function connects the drone to the computer.

44. **<disconnect_to_computer> ::= disconnect()**

This disconnects the drone from the computer. It can only be called when the height is equal to zero.

45. **<turn_on_off_spray> ::= changeSpray(<logic_expr>)**
| changeSpray(<identifier>)

It changes the situation of the spray. When the user enters true, the spray will be opened and when the user enters false, the spray will be closed.

46. **<set_altitude> ::= setAltitude(<arithmetic_exprs>)**

It sets the altitude of the drone and the drone will climb up or down according to current altitude and entered altitude.

47. **<set_heading> ::= setHeading(<arithmetic_exprs>)**

It sets the heading direction. If the user enters bigger than 359, it will be decreased to [0-359] interval.

48. **<get_battery_percentage> ::= getBatteryPerc()**

It allows the user to see the battery percentage of the drone. It returns an integer value.

49. **<set_height> ::= setHeight()**

This function will set the ground level. Because, the drone can only be connected or disconnected from the computer on the ground.

50. **<wait_next_instruction> ::= waitNextInst(<arithmetic_exprs>)**

This function will wait the program before running the next instruction. It gets int or identifier as unit of seconds.

51. **<relational_expr> ::= <gt_op> | <lt_op> | <gte_op> | <lte_op> | <eq_op> | <not_eq_op> | <boolean>**

Relational expression represents greater than relations (<gt>), less than relations (<lt>), greater than or equal relations (<gte>), less than or equal relations (<lte>), equal relations (<eq>) and not equal relations (<not_eq>).

52. <and_out> ::= <or_out> | <and_out> && <or_out>

This statement is used for indicating and (&&) relations. It can take and relations on one side of the && symbol and boolean or relational expressions on the other side. It can take or relations on the one side of the && symbol and boolean or relational expressions on the other side. It can take or relations on the both sides or it can take or relations on the one side and and relations on the other side.

53. <or_out> ::= <and_or_in> | <and_or_in> || <or_out>

This statement is used for indicating or (||) relations. It can take or relations on one side of the || symbol and boolean or relational expressions on the other side. It can take and relations on the one side of the && symbol and boolean or relational expressions on the other side. It can take and relations on the both sides or it can take or relations on the one side and and relations on the other side.

54. <and_or_in> ::= <relational_expr>

This statement is used for indicating booleans or relational expressions (gt, lt, etc.).

55. <boolean> ::= true | false

Boolean is used for indicating true or false. This is used for increasing readability and writability because it is common in the other languages.

56. <gt_op> ::= <arithmetic_exprs> > <arithmetic_exprs>

This statement is used for checking whether the left hand side is greater than the right hand side. It can take arithmetic operations on the both sides or on the one side or alternatively it can take two identifiers on the both sides.

57. <lt_op> ::= <arithmetic_exprs> < <arithmetic_exprs>

This statement is used for checking whether the left hand side is less than the right hand side. It can take arithmetic operations on the both sides or on the one side or alternatively it can take two identifiers on the both sides.

58. <gte_op> ::= <arithmetic_exprs> >= <arithmetic_exprs>

This statement is used for checking whether the left hand side is greater than or equal to the right hand side. It can take arithmetic operations on the both

sides or on the one side or alternatively it can take two identifiers on the both sides.

59. <lte_op> ::= <arithmetic_exprs> <= <arithmetic_exprs>

This statement is used for checking whether the left hand side is less than or equal to the right hand side. It can take arithmetic operations on the both sides or on the one side or alternatively it can take two identifiers on the both sides.

60. <eq_op> ::= <arithmetic_exprs> == <arithmetic_exprs>

This statement is used for checking whether the left hand side is equal to the right hand side. It can take arithmetic operations on the both sides or on the one side or alternatively it can take two identifiers on the both sides.

61. <not_eq_op> ::= <arithmetic_exprs> != <arithmetic_exprs>

This statement is used for checking whether the left hand side is not equal to the right hand side. It can take arithmetic operations on the both sides or on the one side or alternatively it can take two identifiers on the both sides.

62. <general_expr> ::= <prior_expr> | <general_expr> + <prior_expr>
| <general_expr> - <prior_expr>

It is a term for defining general arithmetic expressions. A general expression may be a prior expression, addition of expressions, or subtraction of expressions.

63. <prior_expr> ::= <term> | <prior_expr> * <term>
| <prior_expr> / <term>
| <prior_expr> % <term>

This expression is created for the purpose of indicating the operations that have priority in the operation process. Multiplication, division and finding remainder operations have priority in an operation sequence. Therefore these operations get executed first, and then the other operations get executed.

64. <term> ::= <input_stmt> | <string> | <int> | <double> | <char> | <identifier> | (<func_call>) | <primitive_funcs>

This statement is used for defining possible terms in arithmetic operations. Term may be a constant, identifier, input or a function call result. For example, in the code “y = x + 5;”, each of y, x, and 5 are terms.

65. <increment_expr> ::= <identifier>++ | ++<identifier>

This statement is used for adding 1 to an integer or a double identifier.

66. <decrement_expr> ::= <identifier>-- | --<identifier>

This statement is used for subtracting 1 from an integer or a double identifier.

67. <sign> ::= [+-]

This statement is used for indicating the sign of the integers and doubles.

68. <digit> ::= [0-9]

This statement is used for indicating digits. Integers and doubles consist of these digits.

69. <alphabetic> ::= [a-zA-Z_]

Alphabetic is a collection of letters and an underscore symbol. Letters can be either uppercase or lowercase.

70. <dot> ::= \.

Dot stands for “.” character. It is used in the construction of double numbers.

71. <alphanumeric> ::= <digit> | <alphabetic>
| <alphanumeric><digit>
| <alphanumeric><alphabetic>

Alphanumeric is a collection of characters consisting of alphabetic characters, digits or both alphabetic characters and digits.

72. <identifier> ::= <alphabetic> | <alphabetic><alphanumeric>

An identifier is used to give a name to a variable (e.g., int identifier = 1903;). It has to start with an alphabetic character but then it can have an alphanumeric character.

73. <string> ::= \"([^\\\"]|\\\"|\\\\|\\\n)*\"

A string is a variable type which is used to define a set of characters. A string variable has to be defined in quotation marks (e.g., string str = “Grondstof”;).

74. <int> ::= <sign><digit> | <digit>

An int is a variable type which is used to define integer numbers. It can be either positive or negative.

75. <double> ::= <sign><digit><dot><digit> | <digit><dot><digit> | <dot><digit>

A double is a variable type which is used to define double numbers. It can be either positive or negative.

76. <char> ::= '[^\']'

A char is a variable type which is used to define a single character. A char variable has to be defined in single quotation marks (e.g., char c = 'a').

77. if

"if" is used to indicate if statements. In order to increase readability and writability, the common reserve word for if statements is used.

78. else

"else" is used as a reserved keyword which is used for "if-else" statements. "Else" statement is used in case a condition is not satisfied in the "if" statement.

79. else_if

"else_if" is used as a reserved keyword which is used for "else if" statements. It is used for indicating else if statements.

80. while

"while" is a reserved keyword which is used for declaring while loops. "while" is picked because it is a commonly used word in programming languages.

81. for

"for" is a reserved keyword used for declaring for loops. "for" is picked because it is a commonly used word in programming languages.

82. function

"function" is a reserved keyword used for declaring functions. "function" is picked because it is a commonly used word in programming languages.

83. return

This keyword is used for returning a value from the function. "return" keyword is picked because it is used mostly for returning values.

84. input

This keyword is used for getting input from the user.

85. out

This keyword is used for printing an expression. The difference from the “outln” keyword is that the “out” keyword does not enter a new line in the output console.

86. outln

This keyword is used for printing an expression. The difference from the “out” keyword is that the “outln” keyword shifts to the next line after printing the expression.

87. =

In order to increase readability and writability, the common reserve operator for assignments, equal sign is used to assign variables.

88. ==

Equal equal sign is used to indicate whether the right hand side and left hand side are equal or not.

89. !=

Not equal sign is used to indicate whether the right hand side and left hand side are not equal.

90. +

Plus sign is used to indicate addition operation in order to increase readability, writability.

91. ++

Plus plus sign is used to indicate incrementation by one.

92. -

Minus sign is used to indicate substitution operation.

93. --

Minus minus sign is used to indicate decrementation by one.

94. *

Asterisk symbol is used to indicate multiplication operation. This symbol is commonly used in programming languages, so it is picked to do multiplication.

95. /

Slash symbol is used to indicate division operation. This symbol is picked because it is commonly used in programming languages.

96. %

Percent symbol is used to return the remainder after a number gets divided by another number.

97. <

Smaller than symbol is used for comparing two values. If the left hand side is less than the right hand side, it returns true. If not, it returns false.

98. <=

Smaller than or equal symbol is used for comparing two values. If the left hand side is less than or equal to the right hand side, it returns true. If not, it returns false.

99. >

Greater than symbol is used for comparing two values. If the left hand side is greater than the right hand side, it returns true. If not, it returns false.

100. >=

Greater than or equal symbol is used for comparing two values. If the left hand side is greater than or equal to the right hand side, it returns true. If not, it returns false.

101. ||

This symbol is used for or operation. This symbol is picked because it is commonly used in programming languages.

102. &&

This symbol is used for and operation. This symbol is picked because it is commonly used in programming languages.

103. (

This symbol is used to indicate the left side of parenthesis. A parenthesis is used to determine the priority in logical or mathematical expressions. It is also used in if statements or loops to declare the condition. Therefore, it increases the readability.

104.)

This symbol is used to indicate the right side of parenthesis. A parenthesis is used to determine the priority in logical or mathematical expressions. It is also used in if statements or loops to declare the condition. Therefore, it increases the readability.

105. {

This symbol is used to indicate the left side of curly brackets which is used in if-else statements and loops. It is picked because it is commonly used in programming languages.

106. }

This symbol is used to indicate the right side of curly brackets which is used in if-else statements and loops. It is picked because it is commonly used in programming languages.

107. true / false

These keywords are used to indicate true or false conditions. “true” indicates true conditions and “false” indicates false statements.

3. Evaluation of the Language (Grondstof)

3.1 Readability

We designed our language in order to be read easily by drone users. Therefore, we followed the most common languages’ syntax in most of our BNF. For example, in Grondstof language, statements end with a semicolon (;). If, for and while loops, and function declarations must start with opening curly brace, and end with closing curly brace. Thus, statements inside loops, conditional expressions and functions can be easily determined. In addition, our language supports multi-line comments. User can start a comment by /** and end the comment by **/. All

expressions between these comment indicators are considered as comments. It also increases readability.

For function declarations, users should start with the function type (int, double, string, boolean, char, or nothing) and add “function” reserved keyword after the type. It increases readability because a reader can easily see that it is a function.

A typical Grondstof code starts with the keyword “GRD-START”. Thus, readability increases and users can recognize the start point of the program.

For declaring variables, int, string, boolean, and char keywords are reserved. These keywords are chosen because they are commonly used and easy to understand. A user can understand that the “boolean” keyword stands for true/false.

As a result, people who have a programming background can easily adapt to our language. Even though a person does not know any programming language, Grondstof will be a good choice for starting programming because of increased readability.

3.2 Writability

Writability was one of the main concerns of us while designing Grondstof. As mentioned in the readability section, we aim to use common keywords. There are some improvements compared to other languages. For example, so as to get input from the user, just typing “input();” will be enough. Moreover, to print something, we designed out() and outln() functions. Besides they are easier to write compared to other programming languages’ output statements, they are functional. While outln() prints the expression inside itself and goes to the next line, out() function just prints the expression and does not go to the next line.

Also, if a writer wants to add multi-line comments, our comment blocks will be enough. There is no need to type comment indicators for each line.

There is one more specification that Gronstof has. In order to increment and decrement an identifier by one, users can write “identifier++;”, “++identifier;”, “identifier--;”, and “--identifier;”. This also contributes to writability.

3.3 Reliability

To increase reliability in Grondstof, we define each type with a specific type name (int, double, string, boolean, char). Some languages allow users to declare

every type with one single word such as “var” or “let”. Unlike these languages, Grondstof forces users to use type names in declarations. Thus, reliability increases.

Furthermore, in our arithmetic and logical operations, we consider precedence rules. Multiplication, division and modulus have priority over addition and subtraction. Also, parenthesis provides precedence. For example, if a user types “int x = (3+2)*5;”, addition operation will have precedence because of parenthesis. The result will be 25. We removed the ambiguity.

Finally, we designed our language in such a way that both terminals and nonterminals were used. We went from whole to part. That means, we used nonterminals to define rules and at the smallest pieces, we used terminals. The difference between them is that terminals cannot be splitted into small pieces, whereas nonterminals can.