CS315 - Programming Languages

Project 1 Report

HOVER Language Design

Berk Takıt 21803147 Section-3

Ramazan Melih Diksu 21802361 Section-3

Language Name: HOVER

1.BNF Description of HOVER language

1.Program

<program> ::= <stmt_list>

<stmt_list> ::= <stmt>; | <stmt>; <stmt_list> | <if_stmt>; <stmt_list> | <comment><stmt_list>

<stmt> ::= <declare> | <assign> | <declare_assign> | <expr> | <loop> | <func_call> | <function_declare> | <IO_stmt> | <primitive_func>

<comment> ::= #<string>#

2.Statements

<declare> ::= <type><id_list>

<assign> ::= <id><assignment_op><expr> | <id><assignment_op><func_call>

<declare_assign> ::= <constant_declare_assign> | <type><id> = <expr>

<expr> ::= <id> | <literal> | <arithmetic_expr> | <logical_expr> | <update_expr>

<loop> ::= <for_loop> | <while_loop> | <dowhile_loop> | <times_loop>

<func_call> ::= <id>(<arguments>) | <id>()

<arguments> ::= <id_list> | <literal> | <id_list>,<arguments> | <literal>,<arguments>

<function_declare> ::= <non_void_funtion_declare> | <void_function_declare>

<IO_stmt> ::= <input_stmt> | <output_stmt>

<primitive_func> ::= <inclination> | <altitude> | <temperature> | <acceleration> | <cam_control> | <timestamp> | <wifi> | <storage> | <battery> | <wait>

3.Types

<type> ::= <primitive_type>

<primitive_type> ::= INT | FLT | CHR | BOL | STR

<literal> ::= <signed_int> | <float> | <char> | <string> | NULL

<char> ::= '<all_chars>' | ''

<string> ::= "<string_exp>" | ""

<string_exp> :: <all_chars> | <all_chars><string_exp>

<signed_int> ::= <int> | <negative_int>

<negative_int> ::= -<int>

<int> ::= <digit> | <digit><int>

<float> ::= .<int> | <signed_int>.<int>

<id_list> ::= <id> | <id>,<id_list>

<id> ::= <normal_chars> | <normal_chars><id> | <id><digit>

<all_chars> ::= <normal_chars> | <special_chars> | <space> | <newline>

4.Constants

<constant_declare_assign> ::= CONSTANT <type><id> = <literal>

5.Expressions

<arithmetic_expr> ::= <arithmetic_expr> + <term> | <arithmetic_expr> - <term> | <term>

<term> ::= <term> * <sub_term> | <term> / <sub_term> | <sub_term>

<sub_term> ::= <term> ** <factor> | <term> % <factor> | <factor>

<factor> ::= (<arithmetic_expr>) | <id> | <literal>

<logical_expr> ::= <logical_expr> OR <logic_term> | <logic_term>

<logic_term> ::= <logic_term> AND <logic_factor> | <logic_factor>

<logic_factor> ::= (logical_expr) | !<logical_expr> | <id> | <literal>

<update_expr> ::= <increment> | <decrement>

<increment> ::= <id>++

<decrement> ::= <id>--

<conditional_expr> ::= <id> | <relational_exp> | <range_condition>

<relational_exp> ::= <expr><relational_op><expr>

<range_condition> ::= <id><range_keyword><range_bound>,<range_bound> |
<literal><range_keyword><range_bound>,<range_bound>

<range_bound> ::= <float> | <signed_int> | <id>

<range_keyword> ::= WITHIN | OUTSIDE

6.If Statement

<if_stmt> ::= <matched> | <unmatched>

<matched> ::= IF (<conditional_expr>) {<matched>} ELSE {<matched>} | <non_if_stmt>

<non_if_stmt> ::= <stmt>;<non_if_stmt> | <stmt>;

<unmatched> ::= IF (<conditional_expr>) {<stmt_list>} | IF (<conditional_expr>) {<matched>} ELSE {<unmatched>}

7.Loops

<for_loop> ::= FOR (<for_init>;<for_condition>;<for_update>) {<stmt_list>}

<for_init> ::= <for_init>, <assign> | <for_init>, <declare> | <for_init>, <declare_assign> | <declare_assign> | <assign> | <declare> |

<for_condition> ::= <conditional_expr>

<for_update> ::= <assignment> | <update_expr> |

<while_loop> ::= WHILE (<while_condition>){<stmt_list>}

<times_loop> ::= TIMES (<positive_int>) {<stmt_list>}

<dowhile_loop> ::= DO {<stmt_list>} WHILE (<while_condition>)

<while_condition> ::= <logical_expression> | <id> | <literal> | <conditional_expr>

8.Function Declaration

<non_void_funtion_declare> ::= <type> FUNC <id>(<parameters>){<stmt_list><return_stmt>;} | <type> FUNC <id>(){<stmt_list><return_stmt>;}

<void_function_declare> ::= VOID FUNC <id>(<parameters>){<stmt_list>} | VOID FUNC <id>(){<stmt_list>} | VOID FUNC <id>(<parameters>){<stmt_list>} | VOID FUNC <id>(){<stmt_list>}

<parameters> ::= <parameter> | <parameter> , <parameters>

<parameter> ::= <type> <id>

<return_stmt> ::= RETURN <single_return>

<single_return> ::= <expr> | <func_call> | <single_return><arithmetic_op><expr> | <single_return><arithmetic_op><func_call>

9.IO Statements

<input_stmt> ::= <single_input>

<single_input> ::= READ <id>

<output_stmt> ::= WRITE <expr> | WRITE <conditional_expr>

10.Primitive Functions

<inclination> ::= GET_INC

<altitude> ::= GET_ALT

<temperature> ::= GET_TEMP

<acceleration> ::= GET_ACC

<cam_control> ::= <cam_on> | <cam_off> | <snap_picture>

<cam_on> ::= CAM_ON

<cam_off> ::= CAM_OFF

<snap_picture> ::= SNAP_PIC

<timestamp> ::= GET_TIME

<wifi> ::= CONNECT_BASE

<storage> ::= GET_CAPACITY

<battery> ::= GET_BATTERY

<wait> ::= WAIT <id> | WAIT <positive_int>

11.Operators

<arithmetic_op> ::= + | - | * | ** | / | %

<relational_op> ::= <= | >= | < | > | == | !=

<assignment_op> ::= = | += | -= | *= | /= |**= | %=

12.Symbols

<normal_chars> ::=
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S
|T|U|V|W|X|Y|Z|_

<special_chars> ::= ! | @ | # | \$ | % | ^ | & | * | (|) | + | = | / | * | - | ; | ' | " | ; | '|' | { | } | [|]

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<newline> ::= \n

<space> ::= ' '

<bool> ::= <true> | <false>

<true> ::= TRUE | 1

<false> ::= FALSE | 0

2.Explanation for HOVER Language Constructs

1.<program> ::= <stmt_list>

This non-terminal is the entirety of a single programme, made from a list of statements.

2.<stmt_list> ::= <stmt>; | <stmt>; <stmt_list> | <if_stmt>; <stmt_list> | <comment><stmt_list>

In HOVER, a statement list non-terminal is made out of a combination of statements, if statements and comments. In this non-terminal, we put a semicolon after each type of statement, excluding comments.

3.<stmt> ::= <declare> | <assign> | <declare_assign> | <expr> | <loop> | <func_call> | <function_declare> | <IO_stmt> | <primitive_func>

A single non-if statement non-terminal can be any of the following types of statement: a declare statement, an assign statement, a declare & assign statement, an expression, a loop statement, a function call, a function declaration, an IO statement or a primitive function call.

4.<comment> ::= #<string>#

This non-terminal is used to represent the syntax of a comment in HOVER. By doing this, HOVER language will have a comment functionality where the user can describe anything without interfering with the programme. A comment starts with a hashtag and ends with another hashtag, this way comments can be multi-line with ease.

5.<declare> ::= <type><id_list>

This non-terminal is used for the declaration of the included types in HOVER. HOVER enables the user to declare multiple variables of the same type in a single statement using an id list.

6.<assign> ::= <id><assignment_op><expr> | <id><assignment_op><func_call>

This non-terminal is used to assign expressions to a variable with an assignment operator between. Usage of type allows HOVER to differentiate between chars, strings, ints and floats. These expression types are ids, literals, arithmetic, logical and update expressions. HOVER also enables assigning a return value of a function directly to a variable without an extra copy process.

7.<declare_assign> ::= <constant_declare_assign> | <type><id> = <expr>

This non-terminal serves as a sub-assign unit which allows assignment at the moment of declaration. It also restricts users from declaring a constant without initialization.

8.<expr> ::= <id> | <literal> | <arithmetic_expr> | <logical_expr> | <update_expr>

This non-terminal defines the types of expressions that can be used in HOVER. These types consist of ids, literals plus arithmetic, logical and update operations.

9.<loop> ::= <for_loop> | <while_loop> | <dowhile_loop> | <times_loop>

This non-terminal displays the 4 types of loops that can be used in HOVER, which are: for loop, while loop, do while loop and times loop.

10.<func_call> ::= <id>(<parameters>) | <id>()

This non-terminal is used for function calling. Function calls are made with the id of the function and its parameters given in parentheses. If no parameters exist, right and left parentheses will be written respectively.

11.<arguments> ::= <id_list> | <literal> | <id_list>,<arguments> | <literal>,<arguments>

This non-terminal is used to enable calling a function with the intended amount of ids of variables or literals.

12.<function_declare> ::= <non_void_funtion_declare> | <void_function_declare>

This non-terminal is a sub-declare unit which is specifically used for function declarations. It is separated into 2 types of function declarations: void and non-void function declaration. Due to this specification HOVER is able to perceive the syntax and implementation differences between the two types.

13.<IO_stmt> ::= <input_stmt> | <output_stmt>

This non-terminal represents the input and output statements.

14.<primitive_func> ::= <inclination> | <altitude> | <temperature> | <acceleration> | <cam_control> | <timestamp> | <wifi> | <storage> | <battery> | <wait>

This non-terminal is used for primitive function processes such as: inclination altitude, temperature, camera control, timestamp, wifi, storage, battery and wait.

15.<type> ::= <primitive_type>

This non-terminal represents all the different data types currently in the HOVER language. At the moment there are only primitive types, but in the future more advanced types can be added to extend the language.

16.<primitive_type> ::= INT | FLT | CHR | BOL

The non-terminal primitive_type can take one of the terminal values of INT, FLT, CHR or BOL: representing the data types of integer, float, character and boolean respectively.

17.<literal> ::= <signed_int> | <float> | <char> | <string> | NULL

The non-terminal literal can either be one of the non-terminals signed_int, float, char or string; or it can be the terminal NULL, which represents the NULL value.

18.<char> ::= '<all_chars>' | ''

This non terminal represents a character data type, it can either be any one character between the char identifier ‘ or it can be an empty char ‘’.

19.<string> ::= "<string_exp>" | ""

This non terminal represents a string data type, it can either be a string expression between the string identifier “ or it can be an empty string “”.

20.<string_exp> :: <all_chars> | <all_chars><string_exp>

A string expression can either be a single all_chars non-terminal or it can be any number of all_chars non-terminals put together by using recursion.

21.<signed_int> ::= <int> | <negative_int>

A signed_int non-terminal is used to represent any integer, negative or positive.

22.<negative_int> ::= -<int>

A negative_int non-terminal represents a negative integer, with a - sign at the front of it.

23.<int> ::= <digit> | <digit><int>

The int non-terminal represents a positive integer number. It can either be a single digit or any number of digits by recursion.

24.<float> ::= .<int> | <signed_int>.<int>

The float non-terminal represents a floating point number. It can either be a dot follower by a positive integer, or a signed integer followed by a dot followed by a positive integer.

25.<id_list> ::= <id> | <id>,<id_list>

An id_list non-terminal represents either a single id, or any number of ids separated by commas. This is useful so that the user can declare any number of variables at once.

26.<id> ::= <normal_chars> | <normal_chars><id> | <id><digit>

In HOVER a variable, or a constant, is represented by an id non-terminal. An id has to start with an alphabetical character or ‘_’ but after that it can be any combination of alphanumeric and ‘_’ characters.

27.<all_chars> ::= <normal_chars> | <special_chars> | <space> | <newline>

The all_chars non-terminal represents all the characters a char variable-literal can be. It comprises the alphanumeric characters, special characters, the space character and the newline character.

28.<constant_declare_assign> ::= CONSTANT <type><id> = <literal>

This non-terminal is used for constant declaring. In HOVER constants must have types and specifically tagged with CONSTANT terminal. Furthermore, constants in HOVER must be initialised with a literal. Initializing a constant with another variable's id is not allowed.

29.<arithmetic_expr> ::= <arithmetic_expr> + <term> | <arithmetic_expr> - <term> | <term>

The arithmetic_expr non-terminal represents an arithmetic expression. The hierarchy of this and the following three non-terminals allow us to control the operator precedence. The operator precedence in HOVER goes as follows, any arithmetic expression inside of parentheses is given the highest precedence, aside from that the power operator (**) and the modulo operator(%) have the highest precedence, followed by the multiplication(*) and division(/) operators, and finally the summation(+) and difference(-) operators have the lowest precedence. In the event of a tie, the expression that is to the left is given precedence.

30.<term> ::= <term> * <sub_term> | <term> / <sub_term> | <sub_term>

The term non-terminal is used in creating operator precedence, as described in the arithmetic_expr non-terminal.

31.<sub_term> ::= <term> ** <factor> | <term> % <factor> | <factor>

The term non-terminal is used in creating operator precedence, as described in the arithmetic_expr non-terminal.

32.<factor> ::= (<arithmetic_expr>) | <id> | <literal>

The factor non-terminal is used in creating operator precedence, as described in the arithmetic_expr non-terminal.

33.<logical_expr> ::= <logical_expr> OR <logic_term> | <logic_term>

The logical_expr non-terminal is used to represent a logical expression. The hierarchy of this and the following two non-terminals are used to create operator precedence for logical operators. The precedence, from highest to lowest is as follows: any expression inside parentheses, the not operator (!), the logical and operator (AND), the logical or operator (OR).

34.<logic_term> ::= <logic_term> AND <logic_factor> | <logic_factor>

The logic_term non-terminal is used in creating operator precedence, as described in the logical_expr non-terminal.

35.<logic_factor> ::= (logical_expr) | !<logical_expr> | <id> | <literal>

The logic_factor non-terminal is used in creating operator precedence, as described in the logical_expr non-terminal.

36.<update_expr> ::= <increment> | <decrement>

The update_expr non-terminal is used to represent the increment and decrement operations.

37.<increment> ::= <id>++

The increment non-terminal is used to describe an increment operation using the increment operator ++. In HOVER the only way to use the increment operator is after the identifier, meaning ++x is not a possible usage.

38.<decrement> ::= <id>--

The increment non-terminal is used to describe a decrement operation using the decrement operator -. In HOVER the only way to use the decrement operator is after the identifier, meaning --x is not a possible usage.

39.<conditional_expr> ::= <id> | <relational_exp> | <range_condition>

The condtional_expr non-terminal is used to check for a condition. This can simply be an identifier, a relational experiment such as x > 4, or a range_condition non-terminal which is a special feature of the HOVER language.

40.<relational_exp> ::= <expr><relational_op><expr>

A relational expression in HOVER consists of two expressions with a relational operator between them. Since an expression can be either a logical or arithmetic expression this can lead to relational expressions such as (x AND y) < (1 + 2), which can seem absurd but since boolean values also have an integer value, these expressions are fine.

41.<range_condition> ::= <id><range_keyword><range_bound>,<range_bound> | <literal><range_keyword><range_bound>,<range_bound>

A range condition is one of the special features of the HOVER language. This expression consist of an identifier or literal, followed by one of the words WITHIN or OUTSIDE, and a comma separated duo of bounds. This is simply a shorthand for an expression such as (x < 10 AND x > 0), which by using this new construct can be written as (x WITHIN 0,10), increasing readability and ease of use.

42.<range_bound> ::= <float> | <signed_int> | <id>

A range_bound non-terminal is used to represent one of the bounds in a range statement. A bound can either be an identifier, a signed integer or a float.

43.<range_keyword> ::= WITHIN | OUTSIDE

The range_keyword non-terminal specifies the functionality of a range statement. The WITHIN terminal is used when an inclusion is checked, for example x WITHIN 0,10 returns TRUE when x is in the range [0,10]. The OUTSIDE keyword is the opposite, the x OUTSIDE 0,10 returns TRUE when x is in the range (-inf,0) OR (10,+inf).

44.<if_stmt> ::= <matched> | <unmatched>

This non-terminal is defined to show the types of if statements. It is implemented in order to prevent HOVER from falling into ambiguous categories.

45.<matched> ::= IF (<conditional_expr>) {<matched>} ELSE {<matched>} | <non_if_stmt>

This non-terminal is used to describe the syntax of the nested if statements. The terminals IF and ELSE will be used for clarification reasons. IF terminal will be followed by a conditional expression given in between parentheses. Recursion is used to keep the amount of left and right curly brackets balanced. Number of left and right curly brackets will be kept equal and an if statement can be used in another if statement. A non-if statement can also be inside an if statement.

46.<non_if_stmt> ::= <stmt>;<non_if_stmt> | <stmt>;

This non-terminal is used to show that non-if statements can be singular or followed by other non-if statements.

47.<unmatched> ::= IF (<conditional_expr>) {<stmt_list>} | IF (<conditional_expr>) {<matched>} ELSE {<unmatched>}

This non-terminal is used to describe the syntax of the independent consecutive if statements. The terminals IF and ELSE will be used for clarification reasons. IF terminal will be followed by a conditional expression given in between parentheses. Recursion is used to line up if statements that are independent of each other. Equality of the number of left and right curly brackets will be ignored. Inner implementations will be statement lists between curly brackets for both IF and ELSE terminals. Any of the consecutive if statements can have matched if statements within itself or can be followed by an else statement.

48.<for_loop> ::= FOR(<for_init>;<for_condition>;<for_update>) {<stmt_list>}

In HOVER, a for loop consists of the keyword FOR followed by three specific statements separated by semicolons inside parentheses and a list of statements inside curly braces.

49.<for_init> ::= <for_init>, <assign> | <for_init>, <declare> | <for_init>, <declare_assign> | <declare_assign> | <assign> | <declare> |

A for loop initialization statement can either be a single assignment, a single declaration, a single assignment and declaration or a comma separated list of a combination of these. It can also be empty.

50.<for_condition> ::= <conditional_expr>

A for_condition non-terminal is used to control a for loop.

51.<for_update> ::= <assignment> | <update_expr> |

A for_update terminal is used to update variables. It can either be an assignment operation or an update expression (x++ or x--). It can also be empty.

52.<while_loop> ::= WHILE(<while_condition>){<stmt_list>}

A while loop in HOVER consists of the WHILE keyword, followed by a while condition expression inside parentheses followed by a statement list inside curly braces.

53.<times_loop> ::= TIMES (<positive_int>) {<stmt_list>} | TIMES (<id>) {<stmt_list>}

A times loop is a special feature of the HOVER language, consisting of the keyword TIMES, followed by a positive integer or identifier inside parentheses, followed by a statement list inside curly braces. This construct allows for the shortening of a simple for loop such as for(int x = 0; x< 10; x++){}, which can be written as a times loop as TIMES(10){}. This is used to increase the readability of simple for loops.

54.<dowhile_loop> ::= DO {<stmt_list>} WHILE(<while_condition>)

A do while loop in HOVER is the DO keyword, followed by a list of statements inside curly braces, followed by the WHILE keyword and finally the while condition expression inside

55.<while_condition> ::= <logical_expression> | <id> | <literal> | <conditional_expr>

A while condition non-terminal is used to represent how a while or do while loop can be controlled: by using a logical expression, a conditional expression, an identifier, or a literal.

56.<non_void_funtion_declare> ::= <type> FUNC <id>(<parameters>){<stmt_list><return_stmt>;} | <type> FUNC <id>(){<stmt_list><return_stmt>;}

This non-terminal is used to define the syntax of a non-void function declaration. Type of the function is specified in order to clarify the return type. FUNC terminal is used to improve the readability of HOVER, showing that the given statement is a function. An id for the function must be specified. It should be followed with parentheses that include the parameters if the function has any. If not, parentheses will be written with nothing between. The inner implementation of the function will be done between curly brackets. Inner implementation will include a statement list and a return statement which will end with the terminal value ";".

57.<void_function_declare> ::= VOID FUNC <id>(<parameters>){<stmt_list>} | VOID FUNC <id>(){<stmt_list>}

This non-terminal is used to define the syntax of a void function declaration. Type of the function is not necessary since the return statement does not exist. VOID FUNC terminal is used to improve the readability of HOVER, showing that the given statement is a void function. An id for the function must be specified. It should be followed with parentheses that include the

parameters if the function has any. If not, parentheses will be written with nothing between. The inner implementation of the function will be done between curly brackets. Inner implementation will include a statement list only which will end with the terminal value ";".

58.<parameters> ::= <parameter> | <parameter> , <parameters>

This non-terminal is used to show that a function can have a single parameter or more. If there is more than one parameter , a comma is used to separate them.

59.<parameter> ::= <type> <id>

This non-terminal is used to describe the syntax of a parameter. A parameter must have a certain type and an id.

60.<return_stmt> ::= RETURN <single_return>

This non-terminal describes the syntax of a return statement. RETURN terminal is used followed by the returning element.

61.<single_return> ::= <expr> | <func_call> | <single_return><arithmetic_op><expr> | <single_return><arithmetic_op><func_call>

This non-terminal is used for branching the possible types of the returning element. Returning element can be an expression which means that it can be: id and literal plus arithmetic, logical and update operations. Since the returning element can be literal, the function can return null. A function can also return a function call statement which allows HOVER users to implement recursive programs.

62.<input_stmt> ::= <single_input>

An input statement non-terminal is used to represent an IO read operation. Currently HOVER only supports reading one value at a time but in the future may support reading into multiple variables at once.

63.<single_input> ::= READ <id>

A single input read statement reads a single value into a variable. For example READ x will take an input from stdin and write the value to x.

64.<output_stmt> ::= WRITE <expr> | WRITE <conditional_expr>

An output statement in HOVER consists of the WRITE keyword followed by either an expression or a conditional expression. It writes to the stdout.

65.<inclination> ::= GET_INC

This non-terminal is used to show that the primitive function for getting the inclination is called with the terminal GET_INC. GET_INC will return the inclination of the drone at that exact moment. For example it can be used as a right hand side value during assignments or as an operand.

66.<altitude> ::= GET_ALT

This non-terminal is used to show that the primitive function for getting the altitude is called with the terminal GET_ALT. GET_ALT will return the altitude of the drone at that exact moment. For example it can be used as a right hand side value during assignments or as an operand.

67.<temperature> ::= GET_TEMP

This non-terminal is used to show that the primitive function for getting temperature is called with the terminal GET_TEMP. GET_TEMP will return the temperature of the drone at that exact moment. For example it can be used as a right hand side value during assignments or as an operand.

68.<acceleration> ::= GET_ACC

This non-terminal is used to show that the primitive function for acceleration is called with the terminal GET_ACC. GET_ACC will return the acceleration of the drone at that exact moment. For example it can be used as a right hand side value during assignments or as an operand.

69.<cam_control> ::= <cam_on> | <cam_off> | <snap_picture>

This non-terminal is used to designate the commands for the drone's camera. These commands are cam on, cam off, snap picture.

70.<cam_on> ::= CAM_ON

The cam_on non-terminal is used to define the terminal CAM_ON for the syntax of this function call. This function will initiate the video recording of the drone.

71.<cam_off> ::= CAM_OFF

The cam_off non-terminal is used to define the terminal CAM_OF for the syntax of this function call. This function will finish the ongoing video recording of the drone.

72.<snap_picture> ::= SNAP_PIC

The snap_picture non-terminal is used to define the terminal SNAP_PIC for the syntax of this function call. This function will command the drone's camera to take a picture.

73.<timestamp> ::= GET_TIME

This non-terminal defines the tag for this function call. GET_TIME terminal is used to obtain the current timestamp.

74.<wifi> ::= CONNECT_BASE

The wifi non-terminal is used to connect the drone to the wifi of the base by using the CONNECT_BASE terminal.

75.<storage> ::= GET_CAPACITY

The storage non-terminal is used to get the amount of remaining storage of the drone by using the GET_CAPACITY terminal.

76.<battery> ::= GET_BATTERY

The battery non-terminal is used to get the amount of remaining battery of the drone by using the GET_BATTERY terminal.

77.<wait> ::= WAIT <id> | WAIT <positive_int>

The wait non-terminal is used to put the execution of the next line on hold for a specific amount of time by using the WAIT terminal followed by either a positive integer or an identifier.

78.<arithmetic_op> ::= + | - | * | ** | / | %

The arithmetic_op non-terminal represents one of the arithmetic operator terminals.

79.<relational_op> ::= <= | >= | < | > | == | !=

The relational_op non-terminal is used to represent a relational operator. In HOVER these operators are smaller or equal, bigger or equal, smaller, bigger, equal or not equal, respectively.

80.<assignment_op> ::= = | += | -= | *= | /= |= | %=**

The assignment_op non-terminal is used to represent all the assignment operators. In HOVER you can also shorten arithmetic expressions by using the assignment operators. For example, x = x**2 becomes x **= 2, x = x % 2 becomes x %= 2.

81.<normal_chars> ::=
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_

The normal_chars non-terminal shows the characters that can be used in an id.

82.<special_chars> ::= ! | @ | # | \$ | % | ^ | & | * | (|) | + | = | / | * | - | ; | ' | " | ; | '|' | { | } | [|]

The special_chars non-terminal shows the characters that can also be used in strings or chars other than the normal_chars.

83.<digit> ::= 0|1|2|3|4|5|6|7|8|9

The digit non_terminal shows the digits that can be used in a HOVER language.

84.<newline> ::= \n

The newline terminal shows how to move a one line below using the terminal \n.

85.<space> ::= ' '

The space non-terminal is used to leave a single space.

86.<bool> ::= <true> | <false>

The bool non-terminal represents the boolean data-type and it can either be true or false.

87.<true> ::= TRUE | 1

The true non-terminal represents the true state for a boolean. It can either be TRUE or 1.

88.<false> ::= FALSE | 0

The false non-terminal represents the false state for a boolean. It can either be FALSE or 0.

Reserved Tokens of the HOVER Language:

- **INT:** This token is reserved for stating the integer data type.
- **FLT:** This token is reserved for stating the float data type.
- **CHR:** This token is reserved for stating the character data type.
- **BOL:** This token is reserved for stating the boolean data type.
- **STR:** This token is reserved for stating the string data type.
- **CONSTANT:** This token is reserved for declaring constant identifiers.
- **OR:** This token is reserved for the logical or operator.
- **AND:** This token is reserved for the logical and operator.
- **!:** This token is reserved for the logical not operator.
- **WITHIN:** This token is reserved for the range construct.
- **OUTSIDE:** This token is reserved for the range construct.
- **IF:** This token is reserved for stating if statements.
- **ELSE:** This token is reserved for stating else statements.
- **FOR:** This token is reserved for stating for loops.
- **DO:** This token is reserved for stating do while loops. It is used with the WHILE token.
- **WHILE:** This token is reserved for stating while loops.
- **TIMES:** This token is reserved for stating times loops.
- **RETURN:** This token is reserved for stating return statements.
- **READ:** This token is reserved for stating receiving inputs. This is used before taking an input.
- **WRITE:** This token is reserved for stating giving output. This is used before giving an output.
- **FUNC:** This token is reserved for stating function declarations.
- **VOID:** This token reserved for stating void function declarations. It is used with the FUNC token.
- **GET_INC:** This token is reserved for use in calling the get inclination primitive function.
- **GET_ALT:** This token is reserved for use in calling the get altitude primitive function.
- **GET_TEMP:** This token is reserved for use in calling the get temperature primitive function.
- **GET_ACC:** This token is reserved for use in calling the get acceleration primitive function.
- **CAM_ON:** This token is reserved for use in calling the camera recording on primitive function.
- **CAM_OFF:** This token is reserved for use in calling the camera recording off primitive function.
- **SNAP_PIC:** This token is reserved for use in calling the take picture primitive function.
- **GET_TIME:** This token is reserved for use in calling the get time primitive function.
- **CONNECT_BASE:** This token is reserved for use in calling the connect to base primitive function.
- **GET_CAPACITY:** This token is reserved for use in calling the get capacity primitive function.
- **GET_BATTERY:** This token is reserved for use in calling the get battery primitive function.
- **WAIT:** This token is reserved for use in calling the wait primitive function.
- **TRUE**: This token is reserved for stating the true value of a boolean.
- **FALSE:** This token is reserved for stating the false value of a boolean.

3.HOVER Language Evaluation

a.Readability

The HOVER language is overall decently readable. While it has some different expressions that produce the same result, such as x = x + 1, x += 1 and x++, these are still easy to read expressions and are easy to understand at first glance. This is also true for the range and times loop constructs. While a times loop technically accomplishes the same thing as a for loop, it is much easier to read and understand; and while the range construct is simply a shorthand version of a longer conditional statement, it is also easier to understand. For orthogonality, since there are no complicated data types, it is fairly orthogonal. One problem might be arithmetic operations with different data types. This can be solved by type checking and giving errors if two values used in an arithmetic expression are not of the same data type. The HOVER language is strongly and statically typed, so the types of variables are non-mutable after they are declared, this increases readability. For syntax design, HOVER has clearly readable and understandable special keywords, such as INT, FLT, STR. Also, since all special keywords consist of only uppercase letters, it is easy to differentiate them.

b.Writability

Authoring domain of The HOVER language mostly consists of simplistic elements. HOVER mainly uses the traditional C type of syntaxes. It is a union of very basic and a narrow amount of language constructs. Therefore, users will have familiarity with HOVER's simple constructs, making its writing more effortless. These constructs focus on offering features that are efficient and dependable rather than offering exotic and extraordinary implementations. So, in terms of orthogonality, HOVER users will be able to solve complex problems with understandable features. Moreover, HOVER attempts to restrict the user on certain actions, reducing the error possibilities due to basic constructs. In terms of expressivity, since HOVER concentrates on understandability, it lacks of shortening the syntax for the language constructs on certain points. However, while easing the implementation of already existing features it offers a decent amount of abridgment. For example the usage of TIMES loop is rather more simple than using the other type of loops when we are certain about the loop count. In general, HOVER is a considerably writable language where users can play with facilitated familiar tools.

c.Reliability

The HOVER language is type-checked, so for example if a function has an integer parameter, a character cannot be used instead. HOVER currently has no exception handling implemented. HOVER has no aliasing since it does not use any pointers. Since HOVER is fairly readable and writable, thanks to its simplistic nature, it can be easily written and maintained. Since HOVER will use the C compiler to compile, and is easy to learn, it is pretty cost efficient. Overall, HOVER is decently reliable, and if exception handling is implemented in the future, it can be a pretty reliable language.