CS315 - Programming Languages

Project 2 Report

TEAM 2

HOVER Language

Berk Takıt 21803147 Section-3

Ramazan Melih Diksu 21802361 Section-3

Language Name: HOVER

1.BNF Description of HOVER language

1.Program

<program> ::= START <stmt_list> END

<stmt_list> ::= <stmt> | <stmt> <stmt_list>

2.Statements

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

<non_if_stmt> ::= <declare_assign_stmts> | <constant_expr> | <loop> | <function_declare> | <IO_stmt>

<declare_assign_stmts> ::= <declare_stmts> | <assign_stmt>

<declare_stmts> ::= <declare> | <declare_assign>

<assign_stmt> ::= <assign>

<declare> ::= <single_declare>

<single_declare> ::= <type> IDENTIFIER

<assign> ::= IDENTIFIER<assignment_op><constant_expr>

<declare_assign> ::= <constant_declare_assign> | <single_declare> = <constant_expr>

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

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

<arguments> ::= <arg> | <arg>,<arguments>

<arg> ::= <literal> | IDENTIFIER

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

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

<primitive_func> ::= GET_INC | GET_ALT | GET_TEMP | GET_ACC | GET_TIME | CONNECT_BASE | GET_CAPACITY | GET_BATTERY | CAM_ON | CAM_OFF | SNAP_PIC | WAIT IDENTIFIER | WAIT <positive_int>

3.Types

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

<literal> ::= <integer> | FLOAT | CHAR | STRING | BOOLEAN | VOID

<integer> ::= <positive_int> | <negative_int>

<positive_int> ::= +INTEGER | INTEGER

<negative_int> ::= -INTEGER

4.Constants

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

5.Expressions

<constant_expr> ::= <logical_or_expr> | <range_expr>

<logical_or_expr> ::= <logical_and_expr> | <logical_or_expr> OR <logical_and_expr>

<logical_and_expr> ::= <equality_expr> | <logical_and_expr> AND <equality_expr>

<equality_expr> ::= <relational_expr> | <equality_expr> <equality_op> <relational_expr>

<relational_expr> ::= <arithmetic_add_expr> | <relational_expr> <relational_op> <arithmetic_add_expr>

<arithmetic_add_expr> ::= <arithmetic_mult_expr> | <arithmetic_add_expr> + <arithmetic_mult_expr> | <arithmetic_add_expr> - <arithmetic_mult_expr>

<arithmetic_mult_expr> ::= <arithmetic_pow_expr> | <arithmetic_mult_expr> * <arithemetic_pow_expr> | <arithmetic_mult_expr / <arithmetic_pow_expr>

<arithmetic_pow_expr> ::= <update_expr> | <arithmetic_pow_expr> ** <update_expr> | <arithmetic_pow_expr> % <update_expr>

<update_expr> ::= <primary_expr> | <update_expr>++ | <update_expr>--

<primary_expr> ::= IDENTIFIER | <literal> | <func_call> | <primitive_func> | (<expr>)

<expr> ::= <logical_or_expr>

<range_expr> ::= IDENTIFIER<range_keyword><range_bound>,<range_bound> | <literal><range_keyword><range_bound>,<range_bound>

<range_bound> ::= FLOAT | <integer> | IDENTIFIER

<range_keyword> ::= WITHIN | OUTSIDE

6.If Statement

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

<matched> ::= IF (<constant_expr>) <matched> ELSE <matched> | {<if_stmt_list>}

<unmatched> ::= IF (<constant_expr>) <if_stmt> | IF (<constant_expr>) <matched> ELSE <unmatched>

<if_stmt_list> ::= <stmt_list> | <stmt_list> <return_stmt>; | <return_stmt>;

7.Loops

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

<for_init> ::= <declare_assing_stmts> | <for_init>,<declare_assign_stmts>

<for_condition> ::= <constant_expr>

<for_update> ::= <assign_stmt> | <expr>

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

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

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

8.Function Declaration

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

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

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

<parameter> ::= <single_declare>

<return_stmt> ::= RETURN <single_return>

<single_return> ::= <constant_expr>

9.IO Statements

<input_stmt> ::= <single_input>

<single_input> ::= READ (STRING) INTO IDENTIFIER | READ INTO IDENTIFIER

<output_stmt> ::= WRITE (<constant_exp>) INTO (STRING) | WRITE (<constant_expr>)

2.Explanation for HOVER Language Constructs

1.<program> ::= START <stmt_list> END

This non-terminal is the entirety of a single programme, made from a list of statements. A program in HOVER begins with the START statement and ends with the END statement.

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

In HOVER, a statement list non-terminal is made out of statements.

3.<stmt> ::= <if_stmt>; | <non_if_stmt>; | COMMENT

A single statement can either be an if statement, a non-if statement or a comment.

4.<non_if_stmt> ::= <declare_assign_stmts> | <constant_expr> | <loop> | <function_declare> | <IO_stmt>

A non-if statement can be a declare_assign, constant_expr, loop, function_declare or IO statement.

5.<declare_assign_stmts> ::= <declare_stmts> | <assign_stmt>

Declare assign statements can be either declare statements or a direct assign statement.

6.<declare_stmts> ::= <declare> | <declare_assign>

A declare statement can be a declare or a declare and assign statement.

7.<assign_stmt> ::= <assign>

An assign statement can only be an assign statement.

8.<declare> ::= <single_declare>

A declare can only be a single declare statement. Originally, multi declarations such as INT x,y; were possible in the HOVER language but were removed for the sake of simplicity.

9.<single_declare> ::= <type> IDENTIFIER

A declaration in HOVER is a type followed by an identifier.

10.<assign> ::= IDENTIFIER<assignment_op><constant_expr>

An assignment in HOVER is an identifier, followed by an assignment operator, and a constant expression.

11.<declare_assign> ::= <constant_declare_assign> | <single_declare> = <constant_expr>

In HOVER you can declare and assign a variable on the same line. This is also the only way to declare constant values in HOVER.

12.<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.

13.<func_call> ::= IDENTIFIER(<parameters>) | IDENTIFIER()

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.

14.<arguments> ::= <arg> | <arg>,<arguments>

This non-terminal is used to enable calling a function with the intended amount of arguments.

15.<arg> ::= <literal> | IDENTIFIER

In HOVER an argument can either be a literal or an identifier.

16.<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.

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

This non-terminal represents the input and output statements.

18.<primitive_func> ::= GET_INC | GET_ALT | GET_TEMP | GET_ACC | GET_TIME | CONNECT_BASE | GET_CAPACITY | GET_BATTERY | CAM_ON | CAM_OFF | SNAP_PIC | WAIT IDENTIFIER | WAIT <positive_int>

This non-terminal represents all the primitive functions implemented in the HOVER language. All the primitive functions are called by simply writing their respective keyword, apart from the wait function which accepts either a positive integer or an identifier.

19.<type> ::= INT | FLT | CHR | BOL | STR

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

20.<literal> ::= <integer> | FLOAT | CHAR | STRING | BOOLEAN | VOID

The non-terminal literal can either be the non terminal integer, the VOID keyword which represents a null value, or any of the float, character, string or boolean values.

21.<integer> ::= <positive_int> | <negative_int>

An integer can either be positive or negative.

22.<positive_int> ::= +INTEGER | INTEGER

A positive integer may or may not be preceded by a plus sign.

23.<negative_int> ::= -INTEGER

A negative integer must be preceded by a minus sign.

24.<constant_declare_assign> ::= CONSTANT <type> IDENTIFIER = <literal>

In HOVER a constant is declared by using the CONSTANT keyword and immediately assigning it a literal value.

25.<constant_expr> ::= <logical_or_expr> | <range_expr>

A constant expression can either be a logical-or expression or a range expression, which is a special type of expression unique to the HOVER language. The following hierarchy allows for the control of operator precedence. The precedence of the operators, from highest to lowest are as follows: any expression inside parentheses, the increment (++) and decrement (--) operators, the power (**) and modulo (%) operator, the multiplication (*) and division (/) operators, the sum (+) and difference (-) operators, the relational operators, the equality operators, the logical-and (AND) operator, the logical-or (OR) operator.

26.<logical_or_expr> ::= <logical_and_expr> | <logical_or_expr> OR <logical_and_expr>

27.<logical_and_expr> ::= <equality_expr> | <logical_and_expr> AND <equality_expr>

28.<equality_expr> ::= <relational_expr> | <equality_expr> <equality_op> <relational_expr>

29.<relational_expr> ::= <arithmetic_add_expr> | <relational_expr> <relational_op> <arithmetic_add_expr>

30.<arithmetic_add_expr> ::= <arithmetic_mult_expr> | <arithmetic_add_expr> + <arithmetic_mult_expr> | <arithmetic_add_expr> - <arithmetic_mult_expr>

31.<arithmetic_mult_expr> ::= <arithmetic_pow_expr> | <arithmetic_mult_expr> * <arithemetic_pow_expr> | <arithmetic_mult_expr / <arithmetic_pow_expr>

32.<arithmetic_pow_expr> ::= <update_expr> | <arithmetic_pow_expr> ** <update_expr> | <arithmetic_pow_expr> % <update_expr>

33.<update_expr> ::= <primary_expr> | <update_expr>++ | <update_expr>--

34.<primary_expr> ::= IDENTIFIER | <literal> | <func_call> | <primitive_func> | (<expr>)

The primary expression can be an identifier, a literal, a function call, a primitive function, or an expression inside parentheses. This allows for versatility because we can use function calls and primitive functions anywhere that accepts an expression.

35.<expr> ::= <logical_or_expr>

This rule allows for the combination of different kinds of expressions.

36.<range_expr> ::= IDENTIFIER <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 consists 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.

37.<range_bound> ::= FLOAT | <integer> | IDENTIFIER

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.

38.<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).

39.<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.

40.<matched> ::= IF (<constant_expr>) <matched> ELSE <matched> | {<if_stmt_list>}

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 constant expression given in between parentheses. A non-if statement can also be inside an if statement.

41.<unmatched> ::= IF (<constant_expr>) <if_stmt> | IF (<constant_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 constant expression given in between parentheses. Any of the consecutive if statements can have if statements within itself or can be followed by an else statement.

42.<if_stmt_list> ::= <stmt_list> | <stmt_list> <return_stmt>; | <return_stmt>;

If statement list is implemented so that an if statement can include a return statement when used in functions.

43.<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.

44.<for_init> ::= <declare_assign_stmts> | <for_init>, <declare_assign_stmts>

A for loop initialization statement can be a single declare, an assignment or an assignment with declaration. With recursion, the amount of declare or assignment statements can be increased.

45.<for_condition> ::= <conditional_expr>

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

46.<for_update> ::= <assignment> | <expr>

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

47.<while_loop> ::= WHILE(<constant_expr>){<stmt_list>}

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

48.<times_loop> ::= TIMES (<positive_int>) {<stmt_list>} | TIMES IDENTIFIER {<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.

49.<dowhile_loop> ::= DO {<stmt_list>} WHILE(<constant_expr>)

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 constant expression inside

50.<non_void_funtion_declare> ::= <type> FUNC IDENTIFIER(<parameters>){<stmt_list><return_stmt>;} | <type> FUNC IDENTIFIER(){<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 ";".

51.<void_function_declare> ::= VOID FUNC IDENTIFIER(<parameters>){<stmt_list>} | VOID FUNC IDENTIFIER(){<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 ";".

52.<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.

53.<parameter> ::= <single_declare>

This non-terminal is used to describe the syntax of a parameter. A parameter must be a single declaration.

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

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

55.<single_return> ::= <constant_expr>

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

56.<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.

57.<single_input> ::= READ (STRING) INTO IDENTIFIER | READ INTO IDENTIFIER

A single input read statement can read a single value into a variable or it can directly read a value from an existing file if the pathway is given as a string is between parentheses.

58.<output_stmt> ::= WRITE (<constant_exp>) INTO (STRING) | WRITE (<constant_expr>)

An output statement in HOVER consists of the WRITE keyword followed by either a constant expression or in parentheses. It can also write a constant expression into a file if the pathway is given as a string between parentheses.

3.General Description of HOVER

A program in the HOVER language consists of statements and comments between the START and END keywords. All statements end with a semicolon. An empty program ie START END is not a valid program. All variables need to have a type, which must be given when the variable is first declared. All constants need to be declared and immediately assigned a literal value, variables can’t be used to initialize constants. Non-void functions should always have a return statement at the end of the function body. If statements can have a return statement inside, however a loop cannot have a return statement inside of it (unless it is inside an if statement ie. DO{ if(x==1) return 1; } WHILE (TRUE);). A function call can be used inside a return statement, allowing for recursion.

4.Reserved Tokens of the HOVER Language:

- **START:** This token is reserved for initiating the program.
- **END:** This token is reserved for ending the program.
- **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.
- **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.
- **INTO:** This token is reserved for reading an input into a variable.
- **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.

5.Explanation of Unresolved Conflicts in the HOVER Language:

Yacc design for HOVER involves a single shift/reduce conflict. This conflict is left unresolved on purpose to maintain the functionality of calling functions with more than one argument. Yacc handles this conflict by shifting, since this handling correlates with the aim of the implementation, conflict can be ignored.