BİLKENT UNIVERSITY
CS315 Project 2

HEP Language

Group 15

Pelin Çeliksöz	21600850	Section 2
Elif Özer	21602495	Section 1
Hande Sena Yılmaz	21703465	Section 2

Language Name: HEP Language

The Complete BNF Description of HEP Language

1. Program Grammar

```
<program> ::= BEG <stmt_list> END
<stmt_list> ::=
                | < stmt_list> <stmt_set>
<stmt_set> ::= <non_if_stmt>   | <if_stmt>
```

2. Statements

```
<last_stmt> ::= SEMICOLON


<if_stmt> ::= IF LP <logical_expr> RP LB HASHTAG <stmt_list> HASHTAG RB
                | IF LP <logical_expr> RP LB HASHTAG <stmt_list> HASHTAG RB
ELSE LB HASHTAG <stmt_list> HASHTAG RB


<non_if_stmt> ::= <declare_stmt>
                | <do_while_stmt>
                | <while_stmt>
                | <for_stmt>
                | <output_stmt>
                | COMMENT
                | <return_stmt>
                | <function_declare>
                | <drone_stmts>
                | <function_call>
```

3. Declarations

```
<declare_stmt> ::= <var_declare>
                 | <term>

<var_declare> ::=  <term> ASSIGN_OPERATOR <assignment_values>

<assignment_values> ::= <logical_expr>
```

4. Function Declaration, Function Call and Return Statement

```
<function_declare> ::= <term> LP <parameters> RP LB <stmt_list> RB

<function_call> ::= <term> LP <parameters> RP

<parameters> ::= <param_element>
               | <param_element> COMMA <parameters>

<param_element> ::= <term>
                  | <constant>

<return_stmt> ::= RETURN <expressions>
```

5. Loop Statements

```
<for_stmt> ::=
        FOR LP <var_declare> <last_stmt> <logical_expr> <last_stmt>
<var_declare> RP LB <stmt_list> RB

<while_stmt> ::=
                WHILE LP <logical_expr> RP LB <stmt_list> RB
```

<do_while_stmt> ::= DO LB <stmt_list> RB WHILE LP <logical_expr> RP

6. **Other Statements**

<input_stmt> ::= HEPIN LP <input_body> RP

<input_body> ::= <term>

<output_stmt> ::= HEPOUT LP <output_body> RP

```
<output_body> ::= <primary_expr>
                | <primary_expr> PLUS <output_body>
                |
```

<drone_stmts> ::= <primitive_functions>

```
<primitive_functions> ::= <read_incline >
                        |<read_altitude>
                        |<read_temperature>
                        |<read_acceleration>
                        |<on_camera>
                        |<off_camera>
                        |<take_picture>
                        |<reading_time>
                        |<connect_computer>
```

7. **Primitive Functions**

<read_incline> ::= READ_INCLINE LP RP

<read_altitude> ::= READ_ALTITUDE LP RP

```
<read_temperature> ::= READ_TEMPERATURE LP RP

<read_acceleration> ::= READ_ACCELERATION LP RP

<on_camera> ::= ON_CAMERA LP RP

<off_camera> ::= OFF_CAMERA LP RP

<take_picture> ::= TAKE_PICTURE LP RP

<reading_time> ::= READING_TIME LP RP

<connect_computer> ::= CONNECT_COMPUTER LP RP
```

8. Expressions

```
<expressions> ::= <logical_expr>


<logical_expr> ::= <logical_op>


<logical_op> ::= <or>


<or> ::=  <and>
          | <or> OR <and>


<and> ::=  <in_logic_or>
                | <and> AND <in_logic_or>


<in_logic_or> ::= <out_logic_or>
                   | <in_logic_or> INC_OR <out_logic_or>


<out_logic_or> ::= <in_logic_and>
                      | <out_logic_or> XOR <in_logic_and>
```

```
<in_logic_and> ::= <equality_check>
                        | <in_logic_and> INC_AND <equality_check>

<equality_check> ::= <comparator>
                | <equality_check> CHECK_EQUAL <comparator>
                | <equality_check> NOT_EQUAL_OP <comparator>

<comparator> ::= <arithmetic_op>
                | <comparator> LESS_THEN <additive>
                | <comparator> GREATER <additive>
                | <comparator> LOWER_OR_EQ_OP <additive>
                | <comparator> GREATER_OR_EQ_OP <additive>

<arithmetic_op> ::= <additive>

<additive> ::= <multiplicative>
        | <additive> PLUS <multiplicative>
        | <additive> MINUS <multiplicative>

<multiplicative> ::= <primary_expr>
        | <multiplicative> MULTI <primary_expr>
        | <multiplicative> DIVIDE <primary_expr>
        | <multiplicative> REMAIN <primary_expr>


<primary_expr> ::= term
        | LP <expressions> RP
        | <constant>
        | <drone_stmts>
        | <input_stmt>
        | <function_call>
```

9. Terms and Constants

```
<term> ::= <var>

<var> ::= IDENTIFIER

<constant> ::= INTEGER

          | BOOLEAN

          | STR

          | DOUBLE
```

Symbols

```
<digit> ::=  0|1|2|3|4|5|6|7|8|9
<letter> ::= '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'|
<dot> ::= .
<sign> ::= + | -
<char_ident> ::= '
<string_ident> ::= "
ASSIGN_OPERATOR \=
GREATER \>
LESS_THEN \<
CHECK_EQUAL \=\=
GREATER_OR_EQ_OP \>\=
LOWER_OR_EQ_OP \<\=
NOT_EQUAL_OP \!\=
HASHTAG \#
COMMA \,
SEMICOLON \;
OR \|\|
```

AND \&\&

INC_AND \&

INC_OR \|

XOR \^

MINUS \-

PLUS \+

DIVIDE \/

MULTI *

REMAIN \%

Explanation of HEP Language Constructs

<program> ::= BEG <stmt_list> END

This non-terminal indicates that the begin command starts the program and includes a list of statements and program ends with the end command. This the very base representation of the characteristics that the language possesses in extension.

<stmt_list> ::=

| < stmt_list> <stmt_set>

This non-terminal shows the statement list in the language that consists of an empty statement or a list of statements that has more statements.

<stmt_set> ::= <non_if_stmt> | <if_stmt>

This non-terminal is to represent the possible types of statements that the language possesses. It is either an if statement or non-if statement.

<last_stmt> ::= SEMICOLON

This non-terminal seperates operations between the for loop parantheses with a semicolon expression like for(i = 0; i < 10; i = i + 1)

<if_stmt> ::=

IF LP <logical_expr> RP LB HASHTAG <stmt_list> HASHTAG RB

| IF LP <logical_expr> RP LB HASHTAG <stmt_list> HASHTAG RB ELSE LB HASHTAG <stmt_list> HASHTAG RB

This non-terminal is to indicate the representation of an if statement. If statement may or may not have the else part of an if statement.

<non_if_stmt> ::= <declare_stmt> | <do_while_stmt> | <while_stmt> |<for_stmt> | <output_stmt> | COMMENT | <return_stmt> | <function_declare> | <drone_stmts> | <function_call>

This non-terminal is to indicate the representation of the other types of statements other than if statement.

<declare_stmt> ::= <var_declare>

| <term>

This non-if statement is to declare terms of variables.

<var_declare> ::= <term> ASSIGN_OPERATOR <assignment_values>

In the HEP language, variable declaration is done by assigning <assignment_values> to the <term> by using ASSIGN_OPERATOR

<assignment_values> ::= <logical_expr>

In the HEP language, <assignment_values> are logical expressions represented by <logical_expr>.

<function_declare> ::= <term> LP <parameters> RP LB <stmt_list> RB

In the HEP, <function_declare> is to indicate the declaration of a function. The name of the function is determined by <term> , following that, the parameters are specified between parentheses. Between the LB “{” “and RB “}”, <stmt_list> forms the body of the function.

<function_call> ::= <term> LP <parameters> RP

This non-terminal is used to indicate a function call. The function is called with the term and parameters between the parantheses defined in the function declaration.

<parameters> ::= <param_element>

| <param_element> COMMA <parameters>

In the HEP, function parameters represented by <parameters> can be either <param_element> or <param_element> COMMA <parameters> which means that function may have either single parameter or multiple parameters seperated by comma.

<param_element> ::= <term> | <constant>

In the HEP, parameter can be either <term> or <constant>.

<return_stmt> ::= RETURN <expressions>

In the HEP, functions may return expressions. For that purpose, <return_stmt> is defined as RETURN <expressions> .

<for_stmt> ::= FOR LP <var_declare> <last_stmt> <logical_expr> <last_stmt> <var_declare> RP LB <stmt_list> RB

In the HEP language, <for_stmt> is non-if statement and this statement is used for iterations. This non-terminal is to show the syntax rule of a for statement. Inside the parentheses immediately followed by FOR, the variable initialization is made in the <var_declare> and logical expression either returns true or false to determine iteration is done. While it returns true for loop continues to execute statements and when it returns false the loop immediately stops executing the statements and leaves the loop.

<while_stmt> ::= WHILE LP <logical_expr> RP LB <stmt_list> RB

This non-terminal is to show the syntax rule of while statement. Inside the parentheses immediately followed by WHILE, given logical expression between LP "(" and RP ")" either returns true or false. While it returns true while loop continues to execute statements and when it returns false the loop immediately stops executing the statements and leaves the loop.

<do_while_stmt> ::= DO LB <stmt_list> RB WHILE LP <logical_expr> RP

This non-terminal is to show the syntax rule of do while statements. In do_while the loop starts immediately with the statements and the true or false check is made at the end of the loop after WHILE between LP "(" and RP ")" where the logical expression is. If the result of this logical expression is true, the loop continues executing the statements from the beginning of the loop. If the result of the logical expression is false, then it stops executing and leaves the loop

<input_stmt> ::= HEPIN LP <input_body> RP

This non-terminal is the input stream. It is used when an input is accepted from the user. HEPIN indicates that input is taken from the user.

<input_body> ::= <term>

Input body is <term> which means that user enter any <term>.

<output_stmt> ::= HEPOUT LP <output_body> RP

This non-terminal is the output stream. It is used when the output is displayed on the console. The output is to be displayed on the console is indicated between LP "(" and RP ")" .

<output_body> ::=<primary_expr>

| <primary_expr> PLUS <output_body>

|

<output_body> is to indicate which type is displayed at the end of the <output_stmt>. <primary_expr> or <primary_expr> PLUS <output_body> or empty values can be used in the output_body.

<drone_stmts> ::= <primitive_functions>

This non-terminal represents the drone statements drone can carry out. In the HEP, <drone_stmts> indicates the <primitive_functions>.

<primitive_functions> ::= <read_incline > |<read_altitude> |<read_temperature> |<read_acceleration> |<on_camera> |<off_camera> |<take_picture> |<reading_time> |<connect_computer>

This non-terminal is to indicate primitive functions that control the drone actions to fully operate it with desired features.

<read_incline> ::= read_incline()

This non-terminal is a primitive function that returns the incline of the drone.

<read_temperature> ::= read_temperature()

This non-terminal is a primitive function that reads the temperature of the drone and returns it.

<read_acceleration> ::= read_acceleration()

This non-terminal is a primitive function that reads the acceleration of the drone and returns it.

<on_camera> ::= on_camera()

This non-terminal is a primitive function that turns the camera of the drone on.

<off_camera> ::= off_camera()

This non-terminal is a primitive function that turns the camera of the drone off.

<take_picture> ::= take_picture()

This non-terminal is a primitive function that takes the picture from the drone.

<reading_time> ::= reading_time()

This non-terminal is a primitive function that reads the time starting from the fly of the drone.

<connect_computer> ::= connect_computer()

This non-terminal is a primitive function that connects the drone to the computer.

<expressions> ::= <logical_expr>

In the HEP language, all expressions are connected in order to avoid ambiguity problems. Prior expressions are located deeper in the parse tree. logical expressions are the least priors and <expressions> are <logical_expr> (logical expressions)

<logical_expr> ::= <logical_op>

In the HEP, <logical_expr> is directly logical operation represented by <logical_op>.

<logical_op> ::= <or>

<logical_op> is connected with the or statement <or> so that ambiguity problems is solved since or has less precedence.

<or> ::= <and> | <or> OR <and>

In the HEP, the or expression, <or>, is connect to and expression <and>. <and> has more precedence than <or>

<and> ::= <in_logic_or> | <and> AND <in_logic_or>

In the HEP, <and> expression can connect with <in_logic_or> since <in_logic_or> has greater priority than <and>. In this way, ambiguity problems are avoided.

<in_logic_or> ::= <out_logic_or>

| <in_logic_or> INC_OR <out_logic_or>

<out_logic_or> has greater priority than <in_logic_or> so <in_logic_or> can connect with <out_logic_or> so that ambiguity and priority issues are avoided. INC_OR, defined in the lex, is used in the <in_logic_or> .

<out_logic_or> ::= <in_logic_and>

| <out_logic_or> XOR <in_logic_and>

<in_logic_and> has greater priority than <out_logic_or> so <out_logic_or> can connect with <in_logic_and> so that ambiguity and priority issues are avoided. XOR ,defined in the lex, is used in the <out_logic_or> .

<in_logic_and> ::= <equality_check>

| <in_logic_and> INC_AND <equality_check>

<equality_check> has greater priority than <in_logic_and> so <in_logic_and> can connect with <equality_check> so that ambiguity and priority issues are avoided. INC_AND, defined in the lex, is used in the <in_logic_and> .

<equality_check> ::= <comparator>

| <equality_check> CHECK_EQUAL <comparator>

| <equality_check> NOT_EQUAL_OP <comparator>

In the HEP, equality check expression <equality_check> can connect with comparator in order to avoid ambiguity and precedence issues. comparator has greater priority than <equality_check> and less priority than rest. CHECK_EQUAL and NOT_EQUAL_OP is used in equality check expressions to check equality.

<comparator> ::= <arithmetic_op>

| <comparator> LESS_THEN <additive>

| <comparator> GREATER <additive>

| <comparator> LOWER_OR_EQ_OP <additive>

| <comparator> GREATER_OR_EQ_OP <additive>

In the HEP, comparator expression <comparator> can connect with arithmetic operations represented by <arithmetic_op> in order to avoid ambiguity and precedence issues. <arithmetic_op> has greater priority than <comparator> and less priority than rest.

<arithmetic_op> ::= <additive>

In the HEP, other operations are considered as arithmetic and arithmetic operations represented by <arithmetic_op>.

<additive> ::= <multiplicative>

| <additive> PLUS <multiplicative>

| <additive> MINUS <multiplicative>

<multiplicative> has greater priority than <additive> and less then the rest. In the HEP, additive expression <additive> can connect with multiplicative operations represented by <multiplicative> in order to avoid ambiguity and precedence issues. PLUS and MINUS are used in the additive.

<multiplicative> ::= <primary_expr>

| <multiplicative> MULTI <primary_expr>

| <multiplicative> DIVIDE <primary_expr>

| <multiplicative> REMAIN <primary_expr>

In the HEP, multiplicative expression <multiplicative> can connect with primary expressions represented by <primary_expr> in order to avoid ambiguity and precedence issues. MULTI, DIVIDE and REMAIN are tokens used in the <multiplicative> .

<primary_expr> ::= term

| LP <expressions> RP

| <constant>

| <drone_stmts>

| <input_stmt>

| <function_call>

In the HEP, primary expressions represented by <primary_expr> has the greatest priority. <primary_expr> can be term, constant, drone_statement, input_stmt , function call or LP <expressions> RP

<term> ::= <var>

In the HEP, terms represented by <term> are defined by <var>.

<var> ::= IDENTIFIER

In the HEP language, the variables represented by <var> are defined by IDENTIFIER which is defined in the lex.

<constant> ::= INTEGER | BOOLEAN | STR | DOUBLE

In the HEP language, constants represented by <constant> can be INTEGER, BOOLEAN, STR or DOUBLE, that are defined in the lex.

Nontrivial Tokens and Reserve Words of HEP Language

INTEGER: It is a data type which stores 32 bit signed two's complement integer. Defined in the lex as {sign}?{digit}+

DOUBLE: It is a data type which stores 64-bit IEEE 754 floating point. Defined in the lex as {sign}?{digit}*\.{digit}+

STR: It is a sequence of characters. Defined in the lex as \"([^\\\"]|\\\"|\\\n|\\\\)*\"

CHARACTER: It is a single character. Defined in the lex as \'{alphanumeric}\'

BOOLEAN: It is a primitive data type. It is used to store true or false. Defined in the lex as (true|false)

IF: It is used to determine if a certain condition is true then a block of

statement is executed.

ELSE: It is used to determine if a certain condition is false then else statement is executed.

FOR: It is a control flow statement to iterate the given part of the program

multiple times.

DO: It is a control flow statement to execute the given part of the program at

least once. The further execution depends on the boolean condition.

WHILE: It is a control flow statement to execute the given part of the program repeatedly depending on the boolean condition.

HEPIN: It is the input stream of HEP language. It is used for accepting an input

which is given by the user.

HEPOUT: It is the output stream of HEP language. It is used for displaying the output on the screen.

RETURN: It is used for return statements of the predicate functions in HEP

language.

HASHTAG: This token is used in the if statements to increase readibility and indicate the if statement. #

OR: logical operator, defined in the lex, to perform or operation. ||

AND: logical operator, defined in the lex, to perform and operation. &&

INC_OR: logical operator, defined in the lex, to perform in_logic_or operation(bitwise) . |

XOR: logical operator, defined in the lex, to perform xor operation. ^

INC_AND: logical operator, defined in the lex, to perform in_logic_and operation (bitwise). &

CHECK_EQUAL: logical operator, defined in the lex, to perform equality check. ==

NOT_EQUAL_OP: logical operator, defined in the lex, to perform not equality check. !=

LESS_THEN: logical operator, defined in the lex, to perform less then check operation. <

GREATER: logical operator, defined in the lex, to perform greater then check operation. >

LOWER_OR_EQ_OP: logical operator, defined in the lex, to perform less then or equal check operation. <=

GREATER_OR_EQ_OP: logical operator, defined in the lex, to perform greater then or equal check operation. <=

PLUS: +

MINUS: -

MULTI: *

DIVIDE: /

REMAIN: %

BEG: This token is used as reserve word to indicate that program is started. In HEP, programs start with word, begin.

END: This token is used as reserve word to indicate that program is end. In HEP, programs end with word, end.

SEMICOLON: ;

ASSIGN_OPERATOR: This token defined to make assignments. =

LP: (

RP:)

LB: {

RB: }

COMMA: ,

Conflicts:

There is not any conflict in the HEP language.

Evaluation of HEP Language

For HEP Language, our motivation is to design a meaningful and easy language for drones. First, we determine the BNF part of language and explain language constructs. While doing these, we tried to choose meaningful symbols, words and reserve words to make HEP Language relate to various language criteria such as readability, writability and reliability.

a. Readability

Readability of programming languages is an important issue since it affects both writability and reliability. If a program is easy to read, then learning and writing of this language become easier. We try to make HEP Language easy to read so that this language makes sense for programmers. In HEP Language, constructs, reserve words and other non-trivial tokens are determined by meaningful words and symbols. We consider readability in terms of problem domain. For example, we define Drone Statements as <drone_stmts> which consist of <primitive_functions> and that make sense about drones. If the user/programmer reads the test code, s/he can easily understand what program does thanks to meaningful names, constructs and reserved words. In addition,for instance, we define comments which can so that programmers place comments to some part of languages to make explanations.

b. Writability

Readability and writability of the programming languages are related in terms of re-read what is written. For instance, a programmer should re-read or re-write his/her codes again. In this context, we tried to make HEP Language also easy to write. We make declarations easy to declare. For instance, although there are different variable types in the HEP Language, users do not have to write variable types while declaring. If a user wants to declare an integer named "number" , number = 5 is enough to declare it instead of extending declaration such as "int number=5;" .

c. Reliability

Reliability of programming languages is an important issue since it makes programmers comfortable. We tried to make HEP Language reliable in terms of readability, writability and logically. If a programmer gets an error, s/he can find the error due to ease of readability, writability of HEP Language. Finding such errors easily makes HEP Language reliable in terms of fixing errors. In addition, we avoided ambiguity and priority problems to make HEP a reliable language.