Table of Contents

Revised and Augmented BNF Description of L--

```
<entry_point> ::= <pre_main_statements>void
main(){<program>}<post_main_statements>

<pre_main_statements> ::= <empty>

                                 | COMMENT <pre_main_statements>

<post_main_statements> ::= <empty>

                                 | COMMENT <post_main_statements>

<program> ::= <empty> | <stmt_list>

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


<stmt> ::= <matched_stmt> | <unmatched_stmt>


<matched_stmt> ::= if( <expr> ) <matched_stmt> else <matched_stmt> |
<non_if_stmt>


<non_if_stmt> ::= COMMENT

           | <block>

           | <empty>;

           | <assign_stmt>

           | <func_def>

           | <for_stmt>

           | <while_stmt>

           | <void_func_call>


<unmatched_stmt> ::= if( <expr> ) <stmt>

                      | if( <expr> ) <matched_stmt> else
<unmatched_stmt>


<empty> ::=

<block> ::= {<program>}
```

```
<assign_stmt> ::= <simple_assign>
                      | <final_assign>
                      | <conn_assign>
                      | <conn_decl_assign>

<simple_assign> ::= <decl_assign> | <assign>

<decl_assign> ::= <data_type> VARIABLE = <expr>;
<final_assign> ::= final <data_type> VARIABLE = <literal>;
<assign> ::=  VARIABLE = <expr>; | VARIABLE++; | VARIABLE--;
<update_stmt> ::=  VARIABLE = <expr> | VARIABLE++ | VARIABLE--
<conn_decl_assign> ::= conn VARIABLE = <conn_expr>;
<conn_assign> ::=  VARIABLE = <conn_expr>;

<literal> ::= <bool_lit> | INTEGER | STRING_LIT
<bool_lit> ::= true | false

<data_type> ::= int | bool | string

<expr> ::= <non_comp_expr> | <comp_expr>
<non_comp_expr> ::= <non_comp_expr> \| <xor_expr> | <xor_expr>
<xor_expr> ::= <xor_expr> ^ <and_expr> | <and_expr>
<and_expr> ::= <and_expr> & <add_expr> | <add_expr>
<add_expr> ::= <add_expr> + <mult_expr> | <add_expr> - <mult_expr> |
<mult_expr>
<mult_expr> ::= <mult_expr> * <exp_expr> | <mult_expr> / <exp_expr> |
<exp_expr>
<exp_expr> ::= <exp_expr> ** <un_expr> | <un_expr>
<un_expr> ::= <para_expr>++ | <para_expr>-- | +<para_expr> | -
<para_expr> | !<para_expr> | <para_expr>
<para_expr> ::= (<non_comp_expr>) | <sim_expr>
```

```
<sim_expr> ::= VARIABLE | <literal> | <func_ret> | <func_prim_ret>


<func_prim_ret> ::= <bool_prim_ret> | <int_prim_ret> |
<string_prim_ret>

<bool_prim_ret> ::= getState(<non_comp_expr>)

<int_prim_ret> ::= readTemperature(<non_comp_expr>)

                      | readHumidity(<non_comp_expr>)

                      | readAirPressure(<non_comp_expr>)

                      | readAirQuality(<non_comp_expr>)

                      | readLight(<non_comp_expr>)

                      | readSoundLevel(<non_comp_expr>,
<non_comp_expr>)

                      | readUltrasonic(<non_comp_expr>)

                      | readInfrared(<non_comp_expr>)

                      | readGyroX(<non_comp_expr>)

                      | readGyroY(<non_comp_expr>)

                      | readGyroZ(<non_comp_expr>)

                      | readSmoke(<non_comp_expr>)

                      | readGPSLong(<non_comp_expr>)

                      | readGPSLat(<non_comp_expr>)

                      | readTime()

                      | length(<non_comp_expr>)

                      |  VARIABLE.receiveData()


<string_prim_ret> ::= formatTime(<non_comp_expr>)

                           | str(<non_comp_expr>)

                           | formatTemperature(<non_comp_expr>)

                           | formatHumidity(<non_comp_expr>)

                           | formatAirPressure(<non_comp_expr>)

                           | formatAirQuality(<non_comp_expr>)
```

```
                              | formatLight(<non_comp_expr>)
                              | formatSoundLevel(<non_comp_expr>)
                              | formatUltrasonic(<non_comp_expr>)
                              | formatInfrared(<non_comp_expr>)
                              | formatGyroX(<non_comp_expr>)
                              | formatGyroY(<non_comp_expr>)
                              | formatGyroZ(<non_comp_expr>)
                              | formatSmoke(<non_comp_expr>)
                              | formatGPSLong(<non_comp_expr>)
                              | formatGPSLat(<non_comp_expr>)

<comp_expr> ::= <non_comp_expr> == <non_comp_expr>
                | <non_comp_expr> != <non_comp_expr>
                | <non_comp_expr> \>= <non_comp_expr>
                | <non_comp_expr> \<= <non_comp_expr>
                | <non_comp_expr> \< <non_comp_expr>
                | <non_comp_expr> \> <non_comp_expr>
                | (<comp_expr>)

<void_func_call> ::= <func_ret>; | <void_prim_ret>

<void_prim_ret> ::= puts(<non_comp_expr>);
                     | gets(VARIABLE);
                     |  VARIABLE.sendData(<non_comp_expr>);
                     | switchOn(<non_comp_expr>);
                     | switchOff(<non_comp_expr>);
                     |  VARIABLE.close_conn();

<conn_expr> ::=  build_conn(VARIABLE) | build_conn(<non_comp_expr>,
<non_comp_expr>)
```

```
<func_ret> ::=  VARIABLE(<param_call_list>) | VARIABLE(<empty>)

<param_call_list> ::= <param_call> | <param_call_list>,<param_call>

<param_call> ::= <non_comp_expr>


<func_def> ::= <func_data_type> | <func_void>

<func_data_type> ::= <data_type>  VARIABLE(<param_list>)
{<program><return_stmt>}

                           |<data_type>  VARIABLE(<empty>)
{<program><return_stmt>}

<return_stmt> ::= return <expr>;


<func_void> ::= void  VARIABLE(<param_list>) {<program>}

                      | void  VARIABLE(<empty>) {<program>}


<param_list> ::= <param> | <param_list>,<param>

<param> ::= <data_type> VARIABLE


<for_stmt> ::= for(<simple_assign> <expr>; <update_stmt>) <block>


<while_stmt> ::= while(<expr>) <block>
```

Language Constructs Explanation:

<entry_point>: This is the starting point. The user may write statements, the main program, or post-main statements here.

<pre_main_statements>: This may be empty or have comments before the main program.

<post_main_statements>: This may be empty or may have comments and this happens after the main program.

<program>: A program consists of a statement or a statement list.

<stmt_list>: This represents the list of valid statements in the language.

<stmt>: This represents a statement in the language. A statement may be a matched statement or an unmatched system.

<matched_statement>: Matched statements consist of if or non-if statements. No else is allowed to occur after an if this removes ambiguity as we solved this by included else in unmatched statements.

<non_if_stmt>: A non-if statement can be a comment, a block, it may be empty, it may be an assign statement, a function definition, a for statement, a while statement or a void function call.

<unmatched_stmt>: An unmatched statement may be an if statement on its own or it may be an if statement followed by an else. This allows an else to happen after an if.

<empty>: This represents an absence of text.

<block>: This represents a block statement. A block statement consists of starting curly braces, followed by program and ends with an ending curly brace.

<assign_stmt>: This represents all types of assign statements in the language. An assign statement allows a programmer to assign a variable or value to a variable which may be a constant. This also includes constant declaration in the language.

<simple_assign>: This is a simple assign statement which may be a declaration statement or an assign statement.

<decl_assign>: This is an initialization of a variable in a single statement. It requires the programmer to specify the data type of the variable which he is declaring and also assign an expression to it.

<final_assign>: This represents constant declaration in the language. This requires programmer to specify the final before datatype which will be followed by a variable name. The right-hand side of this statement should be a literal.

<assign>: This represents an assign statement for primitive data types in the language. The right-hand side of this statement is an expression. It may also be an increment or decrement performed on a variable.

<update_stmt>: This represents assigning a new value to an already declared variable which will be followed by an expression. This may also be used to increment or decrement already declared variables.

<conn_decl_assign>: This represents a connection initialization in the language. This requires programmer to specify the datatype "conn" before the variable name. The right-hand side of assign statement consists of connection expression.

<conn_assign>: This represents a connection assignment statement in the language. The right-hand side of this assign statement consists of a connection expression.

<literal>: This represents a literal in the language. A literal can either be a Boolean literal, string literal or an integer.

<bool_lit>: This represents a bool literal in the language which can either be "true" or "false".

<data_type>: This represents a data type in the language. The language supports integer, Boolean and string data types.

<expr>: This represents an expression in the language. An expression may be a non-comparison expression or a comparison expression. This has been done to allow a form of precedence in the form of non-comparison expr happening before xor expr which is followed by and expr followed by add expr which is followed by multiplication expr which is followed by un expr which is followed by un expr which is followed by para expr and this is followed by sim expr at the end.

<non_comp_expr>: A non-comparison expression may be another non-comparison OR xor expression or it may be a lone xor expression.

<xor_expr>: This may be a xor expression XOR a and expression or it may be a and expression.

<and_expr>: This is an and expression AND a add expression or it may be just a single add expression.

<add_expr>: This is an add expression + a multiplication expression or it may be an add expression – a mult expression or it may a single multiplication expression.

<mult_expr>: This is a multiplication expression * multiplication expression or it may be a multiplication expression / exp expression or it may be a single exp expression.

<exp_expr>: This is an exp expression **(raise to the power) and un expression or it may be a single un expression.

<un_expr>: This may be a para expression followed by ++(increment) or a para expression followed by –-(decrement) or it may be a para expression which has a + in front of it to assign a positive expression or it may be a para expression which has a – in front of it to specify negative or it may be a para expression which has a ! in front of it to specify a compliment or it may be a single para expression.

<para_expr>: This may be a non-comparison expression in parenthesis or it may be a simple expression.

<sim_expr>: This is a variable or a literal or it may be a function return or a function primitive return.

<func_prim_ret>: This may be a Boolean primitive return or an integer primitive return or a string primitive return.

<bool_prim_ret>: This expression consists of a getState function which will have non-comparison expression as a parameter. To return the state of the buttons i.e. are they on or off.

<int_prim_ret>: This represents an integer returning primitive function. This includes methods for reading data from sensors, reading time from the system, receiving data from a connection, or getting length of a string.

<string_prim_ret>: This represents a string returning primitive function in the language. This includes methods for formatting data received from the sensors, formatting time, and converting an integer to string.

<comp_expr>: This is any sort of expression containing comparison operators such as ==, !=, or >= etc. This requires two non-comparison expression with the operator in between these expressions.

<void_func_call>: This represents a void function call in the language. The void function does not return anything.

<void_prim_ret>: This represents a void primitive function in the language. These methods include sending data to a connection, toggling a switch, closing a connection or printing a string on the terminal.

<conn_expr>: This represents a connection expression. This may be a connection variable or a "build_conn" primitive function call. This is used to build a connection by passing a variable or a non-comparison expr, non-comparison expr as parameters.

<func_ret>: This represents a non-primitive function call in the language. This function call may return an integer, string, bool, or nothing depending on the function definition.

<param_call_list>: This represents the list of parameters while calling a function in the language. This list can be of arbitrary length. The data type of each parameter should match the parameter in corresponding function definition in exact order.

<param_call>: This is a non-comparison expression. Which may be any type of expression aside from comparison expressions.

<func_def>: This represents function definition in the language. The programmer is required to specify the return data type of the method before specifying the name of the method. It could be either an int, string, bool returning function or a function that does not return anything.

<func_data_type>: This represents a function that returns an integer, string or Boolean in the language. The programmer is required to specify the return data type of the method before specifying the name of the method. In the parentheses, the programmer is required to list the parameters of the function.

<return_stmt>: This represents return statement in the language. This statement cannot be used outside function definition. The data type of the return variable should match the data type specified for return in the function definition.

<func_void>: This represents a function that returns nothing in the language. The programmer is required to specify "void" before specifying the name of the method. In the parentheses, the programmer is required to list the parameters of the function.

<param_list>: This represents parameter list while defining a function in the language. This list can be of arbitrary length.

<param>: This represents a single parameter while defining a function in the language. Before the identifier of each parameter, the programmer is required to specify the data type of the parameter.

<for_stmt>: This represents a for loop statement in the language. The for-loop parentheses must contain three statements. First one is initialization statement which should be a simple assignment statement. Second one is the condition statement which should be a Boolean returning expression followed by a semicolon. The third one is an update statement which should be a simple assignment statement. The initialization statement is run before the for-loop statement. The update statement is run after each iteration in for loop statement. The statement or block after for loop statement is executed if the condition statement returns true.

<while_stmt>: This represents a while loop statement in the language. The programmer is required to specify a bool returning expression inside the parentheses after "while" keyword. After the parentheses, a statement or block follows. The statement or block will run till the expression inside parentheses returns false.

Description of Non-Trivial Tokens:

COMMENT: Any number of characters that come after '#'

MAIN: Entry point function of the program

TRUE: The Boolean statement true.

FALSE: The Boolean statement false.

INT: The primitive data type int, will include both positive and negative integers.

BOOL: The primitive data type bool, this will include both true and false.

STRING: The primitive data type string, a string will be composed of one or more characters and symbols.

CONN: The primitive data type conn which defines a connection by using an IP and a port.

VOID: The void statement for functions with no return.

FINAL: This is used to define constant variables.

IF: The if statement

ELSE: The else statement.

FOR: The for loop statement.

WHILE: The While loop statement.

RETURN: The return statement for methods.

PUTS: Primitive function to print line on standard output.

GETS: Primitive function to get standard input.

LENGTH: Primitive function to get the length of a string.

STR: Primitive function to convert a datatype into a string.

GET_STATE: Primitive function which will get the state of actuators (button parameter) in 1 or 0.

SWITCH_ON: Primitive function which will be used to change the state of the button to on.

SWITCH_OFF: Primitive function which will be used to change the state of the button to off.

SEND_DATA: This Primitive function is used to send data to a connection.

RECEIVE_DATA: Primitive function which is used to get data from a connection.

FORMAT_TIME: Primitive function which will convert the int data provided into time format i.e. in seconds, mins which will be returned as a string.

FORMAT_TEMPERATURE: Primitive function which will format the int provided into degree Celsius returned as a string.

FORMAT_HUMIDITY: Primitive function which will change the int format into percentage humidity and return it as string.

FORMAT_AIR_PRESSURE: Primitive function which will change the int provided to return the pressure as pascals in a string.

FORMAT_AIR_QUALITY: Primitive functions to convert int provided into a string describing air quality.

FORMAT_LIGHT: Primitive function to convert the int provided into a string describing light intensity.

FORMAT_SOUND_LEVEL: Primitive function which will convert the int provided into a string describing the sound level.

FORMAT_ULTRASONIC: Primitive function to convert int parameter into string which describes Ultrasonic level.

FORMAT_INFRARED: Primitive function to convert int parameter into string to measure the infrared readings.

FORMAT_GYRO_X: Primitive function to convert the int parameter into a string representing the X coordinate degrees.

FORMAT_GYRO_Y: Primitive function to convert the int parameter into a string representing the Y coordinate degrees.

FORMAT_GYRO_Z: Primitive function to convert the int parameter into a string representing the Z coordinate degrees.

FORMAT_SMOKE: Primitive function to convert int into a string to measure amount of smoke.

FORMAT_GPS_LONG: Primitive function to convert the int parameter into longitude coordinates.

FORMAT_GPS_LAT: Primitive function to convert the int parameter into latitude coordinates.

READ_TIME: Primitive function to read the time and store it as an Int.

READ_TEMPERATURE: Primitive function to read the temperature and store it as an Int.

READ_HUMIDITY: Primitive function to read the humidity and store it as an Int.

READ_AIR_PRESSURE: Primitive function to read the air pressure and store it as an Int.

READ_AIR_QUALITY: Primitive function to read the air quality and store it as an Int.

READ_LIGHT: Primitive function to read the light intensity and store it as an Int.

READ_SOUND_LEVEL: Primitive function to read the sound level for a specific frequency and store it as an Int.

READ_ULTRASONIC: Primitive function to read the ultrasonic amount and store it as an Int.

READ_INFRARED: Primitive function to read the infrared amount and store it as an Int.

READ_GYRO_X: Primitive function to read the X coordinate of gyro meter and store it as an Int.

READ_GYRO_Y: Primitive function to read the Y coordinate of gyro meter and store it as an Int.

READ_GYRO_Z: Primitive function to read the Z coordinate of gyro meter and store it as an Int.

READ_SMOKE: Primitive function to read the smoke levels and store it as an Int.

READ_GPS_LONG: Primitive function to read the longitude value and store it as an Int.

READ_GPS_LAT: Primitive function to read the latitude value and store it as an Int.

BUILD_CONN: Primitive function to create a connection to a certain IP address and port.

CLOSE_CONN: Primitive function to close previously establish connection.

EXP_OP: This is the arithmetic exponential operator.

MULT_OP: This is the arithmetic multiplication operator.

DIV_OP: This is the arithmetic division operator.

AND_OP: This is the logical AND operator.

OR_OP: This is the logical OR operator.

XOR_OP: This is the logical XOR operator.

NOT_OP: This is the logical NOT operator.

GT_OP: This is the greater than operator.

LT_OP: This is the less than operator.

GTE_OP: This is the greater than or equal to operator.

LTE_OP: This is the less than or equal to operator.

EQ_OP: This is the equal to operator.

NEQ_OP: This is the not equal to or inequality operator.

SEMI_COLON: This is placed at the end of a sentence.

DOT: This is used to access the inner functions of the primitive connection data type.

L_PARA: This is the left parenthesis (.

R_PARA: This is the right parenthesis).

L_CURL: This is the left curly bracket {.

R_CURL: This is the right curly bracket }.

COMMA: This is used to separate several variables when passing them as parameters.

ASSIGN_OP: This is the assignment operator.

INC_OP: This is the increment operator.

DEC_OP: This is the decrement operator.

PLUS: This is the plus sign which may be used to define positive, increments or addition.

MINUS: This is the minus sign which may be used to define negative, decrements or subtraction.

INTEGER: An integer can be any number.

VARIABLE: A variable is defined as an alphabet followed by a combination of any number of digits or alphabets or the '_' symbol.

STRING_LIT: This defines what a string can be, which is anything in between two speech marks "". This includes having speech marks in between provided they have a backslash before them for e.g. \" .

Motivation & Constraints:

Readability

While designing our language "L--", we made sure that we make the language as easily understandable as possible. Since some of our targeted users might not be computer engineers, we have merged several features from other programming languages to make it especially easier for those customers. We are operating under the assumption that since they are buying the IOT devices, they have at least some programming language experience.

Most people who have had some basic computer science knowledge they are familiar with either one of these 3 languages: Java, C++, Python. Keeping this in mind we have made a language which is mostly readable by anyone who has some familiarity with either of these languages. These are some of the features of the L-- language which highlight its readability:

Every primitive data type that we declare, we do that by first writing its data type and then the name of the variable and then it equals the value to be fed into the variable. This feature is similar to that of Java and C++ and thus makes recognition of this new language easy and lowers the learning curve a lot.

A constant type is the same as declaring a variable but has a "final" keyword preceding it. As the keyword explains itself this means that the value put in the variable is final and cannot be changed further in the program.

All the primitive methods used are named to be as self-explanatory as possible e.g. readHumidity() or formatTime(). Thus, making a program written in L-- easily understandable by anyone reading it.

Finally, the mathematical symbols used in L-- +, -, *, /, ** and other operators &, |, ! are very generic and are easily understandable by anyone writing in L--.

Writability:

Keeping in mind the ease writability of the Python language, we have aimed to make L-- as writable as possible. Apart from the declaration of variables with their respective datatypes, the rest of the language is pretty writable.

Since L-- supports integer calculations, we have made it a point to include the support for signed integers so the program can be written easily. In some languages, incrementing a number by one has a lot of implementations such as

```
i = i + 1;

i++;

++i;

i += 1;
```

We have included i = i + 1; and i++; and the rest were discarded. This resulted in making a middle ground between readability and writability so that the programmer can quickly write i++; in the loops but does not have to read different kinds of increments in one program.

Overall, the language is pretty much writable with the conventional looping methods and selection statements and thus to write a program, even someone with little knowledge of C or Java can write easily.

Reliability:

Most of our focus while designing the language was towards reliability. Since this is an IoT language, we have made it so that programmers can easily connect with other IoT devices by simply building a connection with the other device by using the "conn" primitive type and providing a simple IPv6 or IPv4 address in the form of a string.

L-- does everything as it is expected to do as the variables that we are defining are type checked properly. A string cannot be entered into an integer variable and vice-versa. This is a very important feature in IoT as many calculations prove to be difficult if a string type is allowed to change to a different type mid program.

One feature in C is that Boolean type comparations such as A && B is processed even if the variables A and B are integers. We have made it so that only Boolean type variables can be used in this situation and thus making L-- more reliable.

Finally, in L-- we support brackets in every kind of equation, whether it be an integer equation or Boolean expression. This feature increases the reliability of the language by a lot, as the programmer feels a lot comfortable when the different parts of the equation are enclosed in parentheses and the equation is understandable and the result given is then the same as the predicted one.

Overall L-- is focused on all 3 (Readability, Writability, Reliability) language criteria to make the language an overall balanced language which is easily readable, writable and very reliable.