2022-2023 Fall

CS 315 - Programming Languages

Project 2 - Report

GS++ Programming Language

Team 18

Borga Haktan Bilen - 22002733 - Section 1
Berk Çakar - 22003021 - Section 1

Table of Contents

BNF Description of the GS++ Programming Language

```
<program_list> ::= <program>
                 | <program> <program_list>
<program> ::= <start_symbol> <stmt_list> <end_symbol>
<stmt_list> ::= <stmt>;
              | <stmt>; <stmt_list>
              | <comment> <stmt_list>
              | <comment>
<stmt> ::= <conditional_stmt>
         | <declaration_stmt>
         | <assignment_stmt>
         | <loop_stmt>
         | <block_stmt>
         | <jump_stmt>
         | <func_definition>
         | <expr>
<conditional_stmt> ::= if (<expr>) <block_stmt>
                     | if (<expr>) <block_stmt> <else_stmt>
                     | if (<expr>) <block_stmt> <elseif_stmt>
                     | if (<expr>) <block_stmt> <elseif_stmt> <else_stmt>
<elseif_stmt> ::= elseif (<expr>) <block_stmt>
                | elseif (<expr>) <block_stmt> <elseif_stmt>
<else_stmt> ::= else <block_stmt>
```

```
<declaration_stmt> ::= <data_type> <identifier_list>
                     | <data_type> <identifier_list> <assignment_op> <expr>
                     | <data_type> <identifier_list> <assignment_op> <assignment_stmt>
<assignment_stmt> ::= <identifier_list> <assignment_op> <assignment_stmt>
                    | <identifier_list> <assignment_op> <expr>
<loop_stmt> ::= while (<expr>) <stmt>
              | do <stmt> while (<expr>)
              | for (<declaration_stmt>; <expr>; <assignment_stmt>) <stmt>
              | for (<assignment_stmt>; <expr>; <assignment_stmt>) <stmt>
              | for (; <expr>; <assignment_stmt>) <stmt>
              | for (<assignment_stmt>; <expr>;) <stmt>
              | for (<declaration_stmt>; <expr>;) <stmt>
              | for (; <expr>;) <stmt>
<block_stmt> ::= begin <stmt_list> end
<jump_stmt> ::= continue | break | return | return <expr>
<expr> ::= <conditional_expr>
<conditional_expr> ::= <or_expr>
<or_expr> ::= <or_expr> <logical_or_op> <and_expr>
            | <and_expr>
<and_expr> ::= <and_expr> <logical_and_op> <equality_expr>
             | <equality_expr>
<equality_expr> ::= <equality_expr> <equality_op> <relational_expr>
                  | <equality_expr> <inequality_op> <relational_expr>
                  | <relational_expr>
```

```
<relational_expr> ::= <relational_expr> <lt_op> <additive_expr>
                    | <relational_expr> <gt_op> <additive_expr>
                    | <relational_expr> <lte_op> <additive_expr>
                    | <relational_expr> <gte_op> <additive_expr>
                    | <additive_expr>
<additive_expr> ::= <additive_expr> <addition_op> <multiplicative_expr>
                  | <additive_expr> <subtraction_op> <multiplicative_expr>
                  | <multiplicative_expr>
<multiplicative_expr> ::= <multiplicative_expr> <multiplication_op> <primary_expr>
                        | <multiplicative_expr> <division_op> <primary_expr>
                        | <primary_expr>
<primary_expr> ::= (<expr>)
                 | <identifier>
                 | <constant>
                 | <func_call>
                 | <sensor_data>
                 | <timestamp>
<identifier_list> ::= <identifier>
                    | <identifier>, <identifier_list>
<identifier> ::= <initial> | <initial> <more>
<initial> ::= <letter> | $ | _
<more> ::= <final> | <more> <final>
<final> ::= <initial> | <digit>
<data_type> ::= int | bool | float | char | string | long
```

```
<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
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sign> ::= + | -
<symbol> ::= # | $ | % | & | [ | \ | ] | ^ | _ | ` | { | | | } | ~ | *
           | ( | ) | + | , | ! | - | . | / | : | ; | < | = | > | ? | @
<start_symbol> ::= <?gs++
<end_symbol> ::= ?>
<string> ::= <character> | <character> <string>
<character> ::= <non_whitespace_character> | <whitespace_character>
<non_whitespace_character> ::= <symbol> | <letter> | <digit>
<whitespace_character> ::=
<constant> ::= <int_const>
             | <float_const>
             | <char_const>
             | <string_const>
             | <bool_const>
<int_const> ::= <unsigned_int_const> | <signed_int_const>
<signed_int_const> ::= <sign> <unsigned_int_const>
<unsigned_int_const> ::= <digit> | <unsigned_int_const> <digit>
<float_const> ::= <signed_int_const> . <unsigned_int_const>
                | <unsigned_int_const> . <unsigned_int_const>
                | . <unsigned_int_const>
<string_const> ::= "<string>" | ""
<char_const> ::= '<character>' | ''
<bool_const> ::= true | false
<comment> ::= /$ <string> $/
<return_type> ::= <data_type> | void
<func_definition> ::= <return_type> <identifier> () <block_stmt>
                    | <return_type> <identifier> (<param_list>) <block_stmt>
<param_list> ::= <param> | <param>, <param_list>
<param> ::= <data_type> <identifier>
<func_call> ::= <switch_controls>
              | <internet_related_calls>
              | <input_output_related_calls>
              | <identifier> ()
              | <identifier> (<arg_list>)
<arg_list> ::= <expr> | <expr>, <arg_list>
```

```
<sensor_data> ::= <temperature>
                | <humidity>
                | <air_pressure>
                | <air_quality>
                | <light_level>
                | <sound_level>
<temperature> ::= temperature
<humidity> ::= humidity
<air_pressure> ::= air_pressure
<air_quality> ::= air_quality
<light_level> ::= light_level
<sound_level> ::= sound_level
<timestamp> ::= timestamp
<switch_controls> ::= <enable_switch>
                    | <disable_switch>
<enable_switch> ::= enable_switch(<digit>)
<disable_switch> ::= disable_switch(<digit>)
<internet_related_calls> ::= <check_connection>
                           | <connect_to_url>
                           | <send_int_to_conn>
                           | <recv_int_from_conn>
<check_connection> ::= check_connection(<expr>)
<connect_to_url> ::= connect_to_url(<expr>, <expr>)
<send_int_to_conn> ::= send_int_to_conn(<expr>, <expr>)
<recv_int_from_conn> ::= recv_int_from_conn(<expr>)
<input_output_related_calls> ::= <input>
                               | <output>
                               | <formatted_output>
<input> ::= input()
<output> ::= output(<expr>)
<formatted_output> ::= outputf(<string_const>, <arg_list>)
<assignment_op> ::= =
<logical_or_op> ::= ||
<logical_and_op> ::= &&
<equality_op> ::= ==
<inequality_op> ::= !=
<lt_op> ::= <
<gt_op> ::= >
<lte_op> ::= <=
<gte_op> ::= >=
<addition_op> ::= +
<subtraction_op> ::= -
<multiplication_op> ::= *
<division_op> ::= /
```

NOTE: The bold characters and words represents the terminals of the GS++ language.

General Description of the Language's Structure and Explanation of Each Language Construct

The beginning of the execution of a program written in GS++ is specified by the <start_symbol> (which is <?gs++). After this symbol the execution begins and there is no main function requirement for our language, like scripting languages (such as Python and Perl).In other words, every GS++ program is executed in a top-to-bottom manner imperatively. However, every function or variable has to be declared before its call. Execution of a program ends with the <end_symbol> (which is ?>). Between the start and end symbols, a developer can use the following language constructs to develop/implement a GS++ program.

• <program_list>

In order to run multiple valid <program> in a single file, GS++ introduces a <program_list> grammar rule. According to that, each file can contain a single <program>, or a <program> followed by a <program_list>. In a valid GS++ <program_list>, there cannot be any type of written text (including language constructs) except newlines in between two <program>.

• <program>

This non-terminal is the start variable of our grammar for the language GS++. Every valid GS++ derivation starts from this non-terminal. Every GS++ program starts with <?gs++ (which is the <start_symbol>) and ends with ?> (which is the <end_symbol>).

• <stmt_list>

For this non-terminal, we have two different possibilities, one only has one statement followed by a semicolon, and the other is right-recursively defined to include more than one statement in which each statement ends with a semicolon. Under this non-terminal we added some production rules for handling comments. Although comments are not technically statements like others, this is a required design choice in order to add comments to the language. Because, the parser needs to locate a grammar rule if we decide to implement a comment feature in the language. Considering the rest of the grammar, <stmt_list> is the most appropriate place to put <comment> non-terminal.

• <stmt>

This non-terminal includes any other kind of statement non-terminals as a general production rule. <conditional_stmt>, <declaration_stmt>, <assignment_stmt>, <loop_stmt>, <block_stmt>, <jump_stmt>, <func_definition>, and <expr> are the statements included in <stmt>. Only those statements can be used on their own in a valid GS++ program (i.e., using a <elseif_stmt> without a <conditional_stmt> is illegal, parser will throw an error).

• <conditional_stmt>

In GS++, we used **if-elseif-else** logic to implement conditional statements. According to that, an if statement can be followed by zero, one or multiple <elseif_stmt>, and an optional <else_stmt> at the end. In order to eliminate the dangling **else** problem, GS++ requires the use of <block_stmt> after **if**, **elseif**, and **else** (i.e., those statements have to start with **begin**, and end with **end** keyword). In that sense, the last **end** keyword followed by a semicolon determines the end of a <conditional_stmt>.

```
/$ some example <conditional_stmt> use cases in GS++ $/
bool foo = true;

/$ example 1 begins $/
if (foo != true) begin
    int a = 1;
    /$ ... $/
end; /$ example 1 ends, see the semicolon $/

if (foo) begin /$ example 2 begins $/
    /$ ... $/
end
elseif ((2 * 3) < 8) begin
    string b = "first elseif";
end
elseif (check_connection(2) == true) begin
    send_int_to_conn(1000, 2);
    /$ ... $/
end; /$ example 2 ends $/

/$ example 3 begins $/
if (foo != true) begin
    int a = 1;
    /$ ... $/
elseif (5 != 3.2) begin
    /$ ... $/
end
else begin
    /$ ... $/
end; /$ example 3 ends $/

else begin /$ or elseif $/
    char c = 'c';
end; /$ syntax error since this else (or elseif) does not belong to an if statement $/
```

• <elseif_stmt>

This non-terminal is designed to be used in <conditional_stmt>. Like in the case of **if** statements, an <elseif_stmt> has to be followed by a <block_stmt> to remove the ambiguity. In a <conditional_stmt>, there may be zero, one, or multiple <elseif_stmt>. Recursive definition of the <elseif_stmt> allows the use of multiple **elseif** in <conditional_stmt>.

• <else_stmt>

An <else_stmt> is the optional last piece of a <conditional_stmt>. In the case that <expr> given in the all available **if** and **elseif** fail, if it exists in the <conditional_stmt>, <else_stmt> will be executed. Similar to **if** and **elseif**, an <else> has to be followed by a <block_stmt> to eliminate the dangling **else** issue.

• <declaration_stmt>

This non-terminal is for declaring variables in the language. Accordingly, GS++ features a few declaration options included in this production rule, shown in the following code section:

```
int x;                  /$ declare x without an initial value $/
int y = 5;              /$ declare y with an initial value $/
int x, y, z, t = 4;     /$ declare x, y, and z and assign them to an initial value $/
/$ declare x with an initial value while assigning the same value $/
/$ to already declared variables y, z and t $/
int x = y = z, t = 4;
```

• <assignment_stmt>

This non-terminal is for assigning values to the variables. For this purpose, the right-hand side of an assignment operator can be either a result of an expression or another <assignment_stmt>. Following code section shows some of these different assigning options:

```
char x, y, z, t;
x = 'c';         /$ assign x to a value $/
y = z, t = 'l';  /$ assign z to a value then assign y to z and t $/
bool t;
t = (y == z);    /$ assign t to the result of an expression $/
```

• <loop_stmt>

For the loops, we followed the conventions from the C, C++, and Java programming languages (in other words, C-family programming languages). Therefore, our language provides while, do-while, and for loops. All loop statements run the <stmt> inside them until the specified logical expression is evaluated as false.

• <block_stmt>
This non-terminal allows to group multiple statements as one statement. Blocks begin with the **begin** keyword and end with the **end** keyword. This **begin-end** notation was inspired by the Pascal programming language.

• <jump_stmt>
This non-terminal has statements that can affect the structural flow of the program. These statements are: **continue**, **break**, **return**. For this non-terminal, we followed the convention from C-family programming languages.

• <expr>
It is a non-terminal for all other kinds of expressions. Expressions are statements that evaluate to a result (i.e., return a value).

• <conditional_expr>, <or_expr>, <and_expr>, <equality_expr>, <relational_expr>, <additive_expr>, <multiplicative_expr> and <primary_expr>
To ensure the operation and operator precedence in GS++, expressions are layered based on their precedence. In our design, conditional expressions which correspond to expressions that include || and && operators in them have the lowest priority since they are at the top of the layered production rules. On the other hand, equality expressions with == and != operators have precedence conditional expressions as we proceed through the deeper layers of expression-related production rules. Similarly, relational expressions which include <, >, <=, and >= operators have the same precedence among themselves. However, they are prioritized compared to the conditional and equality expressions. Furthermore, additive expressions, addition and subtraction, have the same precedence in between their own kind. Despite having precedence over conditional, equality, and relational expressions, additive expressions are not prioritized over multiplicative expressions. Similarly, in multiplicative expressions, multiplication and division have the same precedence. Also, they have prioritized over the additive expressions and the ones additive expressions have precedence over. Lastly, primary expressions such as expressions in parentheses, the result of the function calls, and constants have the most precedence since primary expressions are the last layer of the production rule chain for the expressions. We also considered the associativity for additive and multiplicative expressions (other expressions are non-associative). Since these expressions are required to be left-associative (i.e., operation evaluation flow is from left to right), their production rules are defined as left recursive rules. See the relation below to comprehend the operator precedence rules in GS++.

Lowest Precedence to Highest

<conditional_expr> → <or_expr> → <and_expr> → <equality_expr> → <relational_expr> → <additive_expr> → <multiplicative_expr> → <primary_expr>

• `<primary_expr>`
This non-terminal includes identifiers, constants, function calls, sensor data (IoT-related) and timestamp from the timer of an IoT device, which allows them to be used in any expression. With this design pattern, we provide flexibility to the language.

• `<identifier_list>`
This non-terminal allows statements to either have one identifier or more than one identifier separated by commas using right recursion.

• `<identifier>`
For this non-terminal we followed Java convention which uses `<initial>`, `<more>` and `<final>` for the construction of any identifier (variable or function name). An identifier can only start with an `<initial>` and followed by `<more>`. Identifiers cannot be reserved keywords.

• `<initial>`
This non-terminal defines the beginning characters for an identifier. These characters can be either a capital or non-capital letter and **$** and **_** characters.

• `<more>`
This non-terminal defines the characters that can come after the `<initial>` of an identifier. It is defined as left recursive.

• `<final>`
This non-terminal is for defining alphanumeric characters such that a <final> can be either a digit `[0-9]` or a `<initial>` (which includes letters and **$** and **_** symbols).

• `<data_type>`
This non-terminal defines the primitive data types that can be used in GS++. For now, our language has **`int`**, **`bool`**, **`float`**, **`char`**, **`string`**, and **`long`** data types. **`long`** data type (64-bit) is specially introduced to store timestamp values for the IoT nodes. Because our **`int`** data type is signed and follows the Java programming language convention, epoch timestamps will not fit into 32-bit **`int`** fields after 19.01.2038.

• `<letter>`
Non-terminal to define the capitalized or non-capitalized letters in the English alphabet.

• `<digit>`
This non-terminal defines digits from (including **`0`** and **`9`**) **`0`** to **`9`**.

• <sign>
This non-terminal is evaluated as either **-** or **+.** They are used mainly for signed integer constants.

• <symbol>
This non-terminal defines 30 ASCII standard symbols.

• <start_symbol>
This non-terminal (**<?gs++**) defines the start symbol, which is required to have at the beginning of every valid GS++ program. This notation was influenced by the PHP programming language.

• <end_symbol>
This non-terminal (**?>**) defines the end symbol for every valid GS++ program.

• <string>
This non-terminal defines the string structure using right recursion. Strings consist of characters.

• <character>
This non-terminal can be evaluated either as a non-whitespace character or a whitespace character.

• <non_whitespace_character>
A non whitespace character can be either a <symbol>, <letter> or <digit>

• <whitespace_character>
This non-terminal corresponds to a single regular whitespace character (" ").

• <constant>
This non-terminal is the parent for the <int_const>, <float_const>, <char_const>, <string_const>, and <bool_const> non-terminals.

• <int_const>
Integer constants are the nominal integer values in the set of real numbers. <int_const> production rule can generate infinite number of integer values according to GS++ grammar. Integer constants have two forms which they are; signed (<signed_int_const>) and unsigned (<unsigned_int_const>).

• <signed_int_const>
Signed integer constants are integers that consist of a sign at the beginning followed by an unsigned integer (<unsigned_int_const>).

• <unsigned_int_const>
This non-terminal consists of digits (<digits>), and because this rule is defined using left recursion it can generate an infinite number of unsigned integer values according to our grammar.

• <float_const>
A float constant have three alternatives: A signed integer constant (<signed_int_const>) followed by a dot (**.**) and an unsigned integer constant (<unsigned_int_const>) after the dot, a single dot (**.**) followed by a unsigned integer (<unsigned_int_const>) or an unsigned integer constant (<unsigned_int_const>) followed by a dot (**.**) and another unsigned integer constant (<unsigned_int_const>) just after the dot.

• <string_const>
For strings, we used the common string representation, which is a string (<string> non-terminal) between two double quotes (**"**).

• <char_const>
This non-terminal corresponds to a <character> which is surrounded by **''** (single quotes).

• <bool_const>
This non-terminal covers the one and only boolean literals found in any programming language except C programming language, which are **true** and **false** literals.

• <comment>
Comments in GS++ language are defined as **/$** followed by a <string> and end with the **$/**. Only the line comments are supported. Comments can only appear individually on a line without any other statements, or at the very beginning and/or end of a line with other statements.

• <return_type>
This non-terminal specifies the grammar rule for the function return types by combining <data_type> and **void**, similar to C programming language.

• <func_definition>
This non-terminal specifies the grammar rule for declaring a function. This production rule is similar to the C programming languages for defining a function.

• <param_list>
This non-terminal defines the grammar of the parameters that are taken by the functions of the GS++ language. According to that, <param_list> can consist of a single <param>, or a <param> followed by a comma (**,**) and <param_list>.

• <param>
This non-terminal defines the parameters that are used in the function definition for retrieving data from outside of the function. It is structured similarly to the C-family programming languages, where a data type (<data_type>) is followed by an identifier (<identifier>) (name of the variable) to form a parameter.

• <func_call>
This non-terminal corresponds to the function calls made to the IoT-related language features or user-defined functions.

• <arg_list>
This non-terminal defines the grammar of the arguments that are given to the functions of the GS++ language. According to that, <arg_list> can consist of a single <expr>, or a <expr> followed by a comma (**,**) and <arg_list>.

• <sensor_data>
This non-terminal is the parent of the other non-terminals related to IoT nodes' (in essence, temperature sensor, humidity sensor, air pressure sensor, and more) data.

• <temperature>
This non-terminal is used to define the special keyword **temperature** in the GS++ language, which this keyword is used to read the temperature data from the sensor.

• <humidity>
This non-terminal is used to define the special keyword **humidity** in the GS++ language, which this keyword is used to read the humidity data from the sensor.

• <air_pressure>
This non-terminal is used to define the special keyword **air_pressure** in the GS++ language, which this keyword is used to read the air pressure data from the sensor.

• <air_quality>
This non-terminal is used to define the special keyword **air_quality** in the GS++ language, which this keyword is used to read the air quality data from the sensor.

• <light_level>
This non-terminal is used to define the special keyword **light_level** in the GS++ language, which this keyword is used to read the light level data from the sensor.

• <sound_level>
This non-terminal is used to define the special keyword **sound_level** in GS++ language, which this keyword is used to read the sound level data from the sensor.

• <timestamp>
This non-terminal is used to define the special keyword **timestamp** in GS++ language, which this keyword is used to read the timestamp data from the sensor.

• <switch_controls>
This non-terminal has two alternatives: either it produces <enable_switch> or <disable_switch>, which these two non-terminals are related to the switch operations on the IoT nodes.

• <enable_switch>
This non-terminal defines a built-in language function (**enable_switch(**<digit>**)**) to enable the switch whose ID is given by an user-passed digit (we pass digit to the function because there are only 10 switches [ID's are from 0 to 9]). The aim of enabling switches is to control some actuators. This function is designed to return a <bool_const> based on the success of the trial.

• <disable_switch>
This non-terminal defines a built-in language function (**disable_switch(**<digit>**)**) to disable the switch whose ID is given by an user-passed digit (we pass digit to the function because there are only 10 switches [ID's are from 0 to 9]). The aim of disabling switches is to control some actuators. This function is designed to return a <bool_const> based on the success of the trial.

• <internet_related_calls>
This non-terminal is a parent for the language built-in function non-terminals related to the internet operations (i.e., check the connection, connect to URL, send and/or receive data to/from a connection).

• <check_connection>
This non-terminal defines a built-in language function called **check_connection(**<expr>**),** which is called by the user to check whether the IoT node is connected to the URL with the given ID (ID of the connection to a specific URL, specified during the establishment of the connection by the user) that is passed as an <expr> as an argument. This function is designed to return a <bool_const> based on the availability of the connection.

• <connect_to_url>
This non-terminal defines a language built-in function called **connect_to_url(**<expr>, <expr>**).** This function is called by the user to connect the node to the given URL, given by the <expr> as an first argument and also by assigning an ID to the connection by passing an <expr> as an second argument, user will be able to use a specific connection by using it's unique ID (assigned by user) this will allow the user to connect to multiple URLs. This function is designed to return a <bool_const> based on the success of the URL connection attempt.

• <send_int_to_conn>
This non-terminal defines a language built-in function called **send_int_to_conn(**<expr>, <epxr>**)**. This user-called function is responsible for sending an integer value, which is the result of the <expr> which is the first parameter taken by the function, to a specific connection which is specified by the ID given as an argument that is the second <expr> in the arguments. This function is designed to return a <bool_const> based on the success of the operation.

• <recv_int_from_conn>
This non-terminal defines a language built-in function called **recv_int_from_conn(**<expr>**)**. This user-called function is responsible for receiving an integer value from the connection with the specified ID that is taken in the form of <expr> as a parameter. This function is designed to return a <int_const> based on the integer value received from the connection.

• <input_output_related_calls>
This non-terminal includes the language built-in function non-terminals related to the user input and output operations.

• <input>
This non-terminal defines a language built-in function called with the keyword of **input()**. This function is responsible for receiving inputs from a console/terminal. This function has the same functionality with Python's **input()** function. It will take the input given by the user until the first newline. It is designed to return the value of the user-given input in the type of the left-hand side of the assignment or declaration statement.

• <output>
This non-terminal defines a language built-in function called with the keyword of **output(**<expr>**)**. This function outputs the given expression to the console/terminal by using the result of the given <expr> argument (in essence, its string representation).

• <formatted_output>
This non-terminal defines a language built-in function called with the following keyword, **outputf(**<string_const>**,** <arg_list>**)**. This function formats the string representation of every <expr> in the <arg_list> according to the formatting styles (using conversion specifiers [such as “%d”]) into the given <string_const> (the first passed argument), with respect to order of specifiers and <expr> in the <arg_list>. GS++ uses the same conversion specifiers with the C and C++. The formatted text is going to be printed on the console/terminal, similar to <output>.

• <assignment_op>
This terminal matches the assignment operator (**=**).

• `<logical_or_op>`
This terminal matches the logical or operator (**||**).

• `<logical_and_op>`
This terminal matches the logical and operator (**&&**).

• `<equality_op>`
This terminal matches the equality operator (**==**).

• `<inequality_op>`
This terminal matches the inequality operator (**!=**).
• `<lt_op>`
This terminal matches the less than operator (**<**).

• `<gt_op>`
This terminal matches the greater than operator (**>**).

• `<lte_op>`
This terminal matches the less than or equal to operator (**<=**).

• `<gte_op>`
This terminal matches the greater than or equal to operator (**>=**).

• `<addition_op>`
This terminal matches the addition operator (**+**).

• `<subtraction_op>`
This terminal matches the subtraction operator (**-**).

• `<multiplication_op>`
This terminal matches the multiplication operator (*****).

• `<division_op>`
This terminal matches the division operator (**/**).

Descriptions of Non-Trivial Tokens

Identifiers:

Identifiers in GS++ can only start with a `<letter>`, **$**, or **_**. After the initial character of the identifier, any `<letter>`, `<digit>`, or **$** and **_** characters can be used for the rest of the identifier. As it is suggested, this convention is similar to Java programming language. We chose this convention based on the motivation of increasing the writability of GS++ since many programmers are already familiar with Java. Additionally, we did not allow the usage of any excessive symbols in the identifiers to increase the readability of the GS++ code.

Literals:

In GS++ language grammar literals correspond to expressions named `<constant>`. Depending on the type of the literal, `<constant>` expressions are divided into five different non-terminals.

Integer Literals:

In GS++ programming language, we consider integer literals (`<int_const>`) in two ways; signed and unsigned. Unsigned integer literals are represented with `<unsigned_int_const>`, and they can include any number of digits without any sign at the beginning. On the other hand, signed integer literals (`<signed_int_const>`) start with a `<sign>` followed by an `<unsigned_int_const>`. We followed this structure in order to make the usage of integer literals intuitive, like in the mathematics notation. This aspect increases the readability and writability of the language.

Float Literals:

Similar to integer literals, we followed natural mathematics notation in GS++ programming language. Accordingly, all float literals (`<float_const>`) are required to have a **.** (dot) in the expression. Each literal has three alternatives for the left side of the **.** (dot). These alternatives can be `<signed_int_const>`, `<unsigned_int_const>` and nothing. If the left side of the **.** (dot) is empty then the float literal has a value between [0, 1). However, the right side of the **.** (dot) can only be a `<unsigned_int_const>`. This structure is used in C-family programming languages, and again, it is the same notation used in mathematics. Therefore, this representation of float literals increases the readability and writability of the language.

Character Literals:

Character literals (`<char_const>`) follow the same convention from C-family programming languages. According to that a `<character>` between single quotes (`'<character>'`) forms a character literal. However, exceptionally, two single quotes (`''`) form an empty character literal. Again, since this representation is a well-known and previously practiced one, it increases the readability and writability of the language.

String Literals:

String literals (<string_const>) are defined similarly to character literals as in the C-family programming languages. Hence, a <string> between double quotes ("<string>") forms a string literal. However, exceptionally, two double quotes ("") form an empty string literal. Similar to the previous cases, in GS++, we think that benefiting from the existing programming languages increases the readability and writability of the language.

Boolean Literals:

Like in any other programming except C, GS++ contains two boolean literals (<bool_const>), which are **true** and **false**. We could skip the boolean literals in our language, as C does, but since they increase the readability and writability of the GS++, we decided to include them.

Comments:

For comments, we tried to keep things simple and included only one comment notation which does not support multi-line comments, since capturing multi-line comments can affect the parsing of the language. According to that, all comments start with **/$** and end with **$/.** Because of this simplicity, the readability, and writability of GS++ increase. We also keep the feature multiplicity lower with this structure.

Reserved Words:

In the GS++ programming language, the reserved words (keywords) are tried to be kept simple and similar to commonly used languages like Java in order to make the language more writable and readable. In addition to the ubiquitous keywords, GS++ has some additional keywords specifically reserved for the IoT-related built-in functionalities. All reserved words in the GS++ language are as follows (see the "Explanations for Each Language Construct" section of this report for detailed descriptions):

- **if**
- **elseif**
- **else**
- **while**
- **do**
- **for**
- **begin**
- **end**
- **send_int_to_conn**
- **check_connection**
- **air_quality**
- **input**
- **continue**
- **return**
- **break**
- **int**
- **bool**
- **float**
- **char**
- **timestamp**
- **recv_int_from_conn**
- **enable_switch**
- **air_pressure**
- **output**
- **long**
- **true**
- **false**
- **void**
- **temperature**
- **humidity**
- **string**
- **light_level**
- **connect_to_url**
- **disable_switch**
- **sound_level**
- **outputf**

Evaluation of the GS++ Programming Language

Writability

In GS++, we have several different keywords and mechanisms. When we designed these, we tried to keep them as minimal as possible. Additionally, we also followed many conventions from C-family programming languages. These allow the programmers who write GS++ to write programs more quickly because they will probably be familiar with the conventions and structures (of non-trivials like comments, literals, identifiers, and keywords) from C-family programming languages. We also added some built-in functions and keywords, specifically the IoT nodes, which a programmer can easily use these functions and keywords without thinking about how these work. This creates an abstraction that improves the writability of the GS++. We also tried to make these keywords as intuitive as possible to make them more memorable (for example, to get the temperature data from the IoT node, the programmer should only use the **`temperature`** keyword in the code). Also, for memorability, we tried to keep the feature multiplicity as optimal as possible while providing every necessary operation and statement. This will improve the overall memorability of the language GS++.

Readability

For readability reasons, we tried to keep the feature multiplicity as low as possible while providing the necessary operations and statements. For example, for incrementing, we did not add any conventional increment statements from C-family programming languages (for instance, `x++`). GS++ also has a comment feature that enables the programmers to explain the codes and/or code blocks they wrote, naturally making the GS++ more readable. Another aspect we considered to improve GS++'s readability is the structure of identifiers. Identifiers in GS++ can start with either a letter (capitalized or not) or two characters, which are (**`$`** and **`_`**), followed by any digit, letter, or the two mentioned characters. This structure is very similar to Java, making the identifiers easier to read (and write). In GS++, there are various functions and keywords specific to IoT nodes. When we designed these, we tried to make them as intuitive as possible in order to make them readable. For instance, for getting the air quality data from the sensor of an IoT node programmer just uses the keyword **`air_quality`**, or for using built-in functions for checking the connection to a URL programmer needs to use the solely **`check_connection()`** built-in function call. Another mechanism we considered for readability is the structures of float literals, which is very similar to natural mathematics (such as 0.7, -5.2, .6). This increases the language's readability. Similar to floating point number structure operators used in the GS++ are also intuitively similar to mathematics; for instance, greater than or equal to operator is **`>=`**. Furthermore, instead of curly braces (**`{ }`**) we used **`begin`** and **`end`** keywords to increase the readability of block statements. We inspired this notation from the Pascal programming language.

Reliability

Although this design report of GS++ only specifies the grammar of the programming language (i.e., not the implementation), we decided to include a few features to ensure reliability in the possible implementation of GS++. First of all, we think that loosely typed languages, such as PHP and Python, are not reliable. For example, one can declare `x = 5` as an integer and then assign `x` to a string with `x = "test"` in these languages. To prevent this, we introduced `<data_type>` (type modifiers) so that users can restrict a variable to a specific type. Secondly, GS++ does not have pointers. Therefore, users cannot perform unintentional actions on the memory at all. Also, since pointers are unavailable, aliasing would not be the case. In other words, two pointers (or references) cannot be set to point to the same variable. To illustrate, one can think of references in C++. If `char y = 'r'; char& z = y;` is considered, the programmer might accidentally change `z`, causing `y` to change as well. Moreover, in GS++, IoT-related built-in functions return a `<bool_const>` (**`true`** or **`false`**) value based on the success of the operation. This way, programmers can design and check their programs according to the success of the IoT operations. Also, empty statements and programs are not allowed in GS++. However, instead of using an empty statement, users can declare their intentions with a comment line, which is valid in GS++. For instance, `if (true) begin end;` is illegal whereas `if (true) begin /$ I want to leave here empty $/ end;` is legal in GS++. Lastly, in order to completely eliminate the dangling **`else`** problem, every **`if-elseif-else`** statement has to be followed by a `<block_stmt>`. According to that, a `<conditional_stmt>` ends when the last **`end`** keyword is followed by a semicolon (as explained comprehensively in the "Explanations for Each Language Construct" section).