**X. Specific Rules**

**a. Basic Rules**

**1. Case-sensitive**

GE++ is a case sensitive programming language. Reserved words are case sensitive while the identifiers have specific naming rules.

**2. Program Delimiter**

**{** characterize the start of the program

**}** characterize the end of the program

**3. Statement Separator**

**.** must be appended at the end of each program statement. Also used in for loop

**4. Rules of the Comment**

Program comments are explanatory statements that helps in reading or writing a source code. GE++ supports single line and multi-line comments. All characters available inside any comment is ignored by the GE++ compiler.

* 1. Comments must be found after the statements in the program
  2. Comments can be placed anywhere in the program
  3. Any Printable characters enclosed in ~~ are considered comments
  4. Any Printable characters enclosed in ~~~ are considered block comments

**~~**  characterize the beginning of a single line comment.

**~~~** characterize as the beginning and/or the end of a block comment

**5. Reserved Words**

|  |  |
| --- | --- |
| **Reserved Words** | **Meaning** |
| boolean | an identifier that can only hold a “false” or “true” value |
| break | end the execution of a current loop or a case statement |
| byte | 8 bit unsigned integer (0 to 255) |
| case | block of code in a switch statement |
| char | declare a character variable |
| const | data of a function that do not change within the program |
| continue | bypass iteration of a loop |
| default | default handler in a case statement |
| do | conjunction with while block to create a do-while loop |
| else | conjunction with if to create an if-else statement |
| float | declare a variable that can hold a 32-bit single floating-point number |
| for | used to create a for-loop |
| goto | jump to a different part of the program |
| if | execute code based on the if-condition |
| int | declare a variable that can hold a 32-bit signed two's complement integer |
| return | used to finish the execution of a method and return the given value |
| string | a sequence of characters |
| switch | used in conjunction with case and default to create a switch statement |
| void | used to declare that a method does not return any value |
| while | used to create a while loop |

**6. Reserved Words for literal values**

|  |  |
| --- | --- |
| **Reserved Words** | **Meaning** |
| false | A Boolean literal value. It has a numeric value of 0 |
| null | A reference literal value |
| true | A Boolean literal value. It has a numeric value of 1 |

**7. Other delimiters and separators**

A delimiter is a character that identifies the beginning or the end of a character string (a contiguous sequence of characters).

**(** used for grouping.

**)** used also for grouping. Used to enclosed expressions starting with “(”.

**[** used in array declaration

**]** used in array declaration. Used to enclosed expressions starting with “[“.

**:** used in switch statement.

**b. Identifier/Variables**

Identifiers are names provided by the programmer. These can be assigned to variables, methods, functions, classes etc. to uniquely identify them to the compiler.

Rules for naming an identifier:

1. Identifier must start with a letter. The consecutive characters can be letters (a-z, A-Z), digits (0-9), hyphen and underscore. Other special characters are not allowed.
2. Maximum number of characters in naming identifier is 20. Exceeding the number will generate an error.
3. Identifier must not belong from the reserve words.
4. Identifier must be unique.
5. It cannot include whitespace.
6. It must be declared before it is referred.

Examples:

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| Age | ag e |
| Name | $name |
| no\_pages | no.pages |
| page-to-go | this-identifier\_have-more-than-20 |
|  |  |

**c. Datatypes**

Refers to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in the storage and how the bit pattern stored is interpreted. There are 6 data types:

**boolean** data type for boolean Can only hold a “false” or “true” value

**byte** data type for byte. 8 bit unsigned integer (0 to 255)

**char** data type for character declare a character variable

**int** data type for integer numbers. declare a variable that can hold a 32-bit

signed two's complement integer

**float** data type for float or real numbers. declare a variable that can hold a 32-bit single

floating-point number

**string**  data type for string. a sequence of characters

Rules:

1. All data types should be in LOWERCASE LETTER.
2. If it is not in lowercase letters, failure to abide to the rules will result to syntax error.

**d. Literals**

**1. Boolean – boolean**

A Boolean represents only two possible states: true or false. If 1 is assigned to a boolean variable it is automatically defined as true, else if 0 is assigned to a boolean variable it is automatically defined as false.

**true** boolean valid for true

**false** boolean valid for false

Example

boolean check = 0.

Then check will be equal to **false**.

**2. Byte - byte**

Byte data type is an 8-bit signed two's complement integer. Default value is 0.

Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.

Rules:

1. Negative numbers must start with a negative sign (-).
2. A double beginning in a plus sign (+) is the same with a byte without it. They are considered a positive number.
3. The value of a double must not exceed 255.

Example:

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| 23 | 0.35 |
| 0 | 124.0 |
| -3 | --7 |
| 113 | 300 |

**3. Character – char**

It is used for identifying the data type of the user defined identifier for characters.

Rules:

1. Char values must be enclosed with a single quote.
2. Any printable characters are allowed

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| ‘%’ | “H” |
| ‘L’ | L |

**4. Integer – int**

It is used for identifying the data type of the user defined identifier for integer.An integer data type is a non-decimal number between -2,147,483,648 and 2,147,483,647.

Rules:

1. An integer must have at least one digit
2. An integer must not have a decimal point
3. An integer can be either positive or negative

Example:

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| 1223 | abe |
| 12 | 12.5 |
| -1 | --1 |

**5. Float - float**

A float (floating point number) is a number with a decimal point or a number in exponential form.

Rules:

1. It begins with at least one digit at followed by a period then followed by at least one digit.
2. Leading and trailing zero are not accepted except for a single zero.
3. Negative numbers must start with a negative sign (-).
4. A float beginning in a plus sign (+) is the same with a float without it. They are considered a positive number.
5. A float must have 7 digits or less.

Example:

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| 1.5 | .5 |
| 0.0 | 12. |
| -1 | --1 |
| 111.5 | 1233456.09100 |
| 1.50 | 1.5012455 |

**6. String - string**

It is used for identifying the data type of the user defined identifier for string or group of characters.

Rules:

1. Load the string engine first to be able to use the string functions.
2. A string must be enclosed with a double quote.
3. Any printable characters are allowed.

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| “Hello World” | ‘Hello World’ |
| “Love” | “Love |

**e. Operators**

An **operator** is a character that represents an action, as for example \* is an arithmetic **operator** that represents multiplication.

**1. Assignment Operators**

= assignment operator which assigns the value of its right operand to its left operand

+= Adds the two operands together, and then assign the result of the addition to the

left operand.

-= Subtracts the two operands together, and then assign the result of the addition to

the left operand.

\*= Multiply the two operands together, and then assign the result of the

multiplication to the left operand.

/= Divide the left operand by the right operand, and assign the result of the division

to the left operand.

%= Perform modular division on the two operands, and assign the result of the

division to the left operand.

**2. Arithmetic Operators**

+ Addition (adds)

- Subtraction (subtracts)

\* Multiplication (multiply)

/ Division(divides)

% Modulo (retains remainder

**Increment Operator**

++a (Prefix) adds 1 before the operand is evaluated

a++ (Postfix) adds 1 after the operand is evaluated

**Decrement Operator**

--a (Prefix) subtracts 1 before the operand is evaluated

a-- (Postfix) subtracts 1 after the operand is evaluated

**3. Relational Operators**

== tests its two operands for equality

!= tests its two operands for inequality

< tests if the value is less than

> tests if the value is greater than

<= tests if the value is less than or equal to

>= tests if the value is greater than or equal to

**5. Logical Operators**

&& tests if two expressions are both true

|| tests if at least one of two expressions it true

! used to flip the truth value

**6. Bitwise Logical Operators**

& examines each bit in its two operands, and when two corresponding bits are both

1, the resulting bit is 1

^ examines each bit in its two operands, and when two corresponding bits are both

0, the resulting bit is 0

| examines each bit in its two operands, and when two corresponding bits are

different, the resulting bit is 1

~ Reverse each bit in its operand

**Operator Precedence**

Aside from the PEMDAS (Parenthesis, ~~Exponent~~, Multiplication, Division, Addition and Subtraction) rule in Mathematics, GE++ also has a well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators.

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator | Description | Associativity |
| 1 | a++ a- -  a()  a[] | Suffix/postfix increment and decrement  Function call  Subscript | Left to right |
| 2 | ++a - - a  +a -a  ! ~ | Prefix increment and decrement  Unary plus and minus  Logical NOT and bitwise NOT | Right to left |
| 3 | A\*b  a/b  a%b | Multiplication  Division  Remainder | Left to Right |
| 4 | A+b  a-b | Addition  Subtraction | Left to Right |
| 5 | <  <=  >  >= | Less than  Less than or equal  Greater than  Greater than or equal | Left to Right |
| 6 | ==  =! | Equal to  Not equal to | Left To right |
| 7 | A&b | Bitwise AND | Left to Right |
| 8 | ^ | Bitwise XOR (exclusive or) | Left to Right |
| 9 | | | Bitwise OR (inclusive or) | Left to Right |
| 10 | && | Logical AND | Left to right |
| 11 | || | Logical OT | Left to right |
| 12 | =  +=  -=  \*=  /=  %= | Direct Assignment  Sum  Difference  Product  Quotient  Remainder | Right to left |
| 13 | , | Comma | Left to Right |

**f. Declaration**

**1. Constant Declaration**

A constant holds a value that does not change. A constant declaration specifies the name, data type, and value of the constant and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Rules for Constant Declaration:

1. To declare a constant, the user must use the reserve word **const** followed by a data type, space, identifier and its equivalent value. It is terminated by a period.
2. Constant Declaration can be declared globally and locally.
3. The value of the constant should follow the rules of the literals of int, char, str, and float.
4. Declare constants one at a time only.
5. Global Constant declaration can be done after the loading of engines and before the function. Local Constant Declaration can be done inside the function.

Syntax:

const <data type> identifier = <value>.

Example:

const float pi = 3.1416.

const int grade = 5.

1. **Variable Declaration**

A **variable** is a storage location paired with an associated symbolic name (an *identifier*), which contains some known or unknown quantity of information referred to as a *value.* All the variables that the program is going to used must be declared.

Rules for Variable Declaration.

1. To declare a variable, the statement should start with an identifier followed by a space followed by the value or multiple values.
2. Assigning value for variable can be done. eg. (x=5). Multiple variable can be declared under one datatype.
3. Multiple identifiers on a single data type can also be done and each of the variable should be separate by a comma. Eg. (a,b,c)
4. The value for variables should follow the rules of the literals of int, str, char, and float.
5. Variables can be declared between any location within the function.
6. Variable declaration can be done after or before any other declaration
7. Multiple declaration in the same line with different data type is not allowed.

Syntax

For single declaration

<datatype> identifier.

For multiple declaration

<datatype> identifier1, identifier2, …, identifierN.

For declaring with value

<datatype> identifier1 = <value>.

Example:

int a.

float a, b, c.

int a=5.

1. **Array**

Array is a data structure which allows a collective name to be given to a group of elements which all have the same type. Array can be treated as a list(vector) that contains values and indexes.

Rules for Array Declaration:

1. Load the array engine first to be able to use the array functions inside that engine.
2. A statement should start with the array data type, followed by the identifier and the size enclosed by a pair of brackets.
3. Comma is being used to separate multiple arrays.
4. Initializing an array whose elements have different data types is not allowed.
5. The first element will be stored in index 0 and the next elements will be stored consecutively.
6. For declaring a 2-dimensional array, the identifier must be enclosed by a pair of brackets.
7. Mathematical Expressions, literals and identifiers in an array index is allowed.

Syntax:

<data type> identifier[<size>].

<data type> identifier[<size>][<size>].

<data type> identifier[<size>], identifier[<size>].

Rules for Initializing Array:

1. Initializing elements of array is done through assigning of elements on it.
2. The elements to be initialized must match with the data type of array.
3. Initializing of array as negative numbers, null or zero is not allowed.
4. The elements must not exceed the array size that is indicated.
5. Elements should be enclosed by curly braces {}.
6. Array index will start at 0.
7. If the initialization of the range of array is less than its length, the rest will be null.

Syntax

<datatype> identifier [<size>] = {<element1>, <element2>, …, <elementN>}.

Rules for Initializing Multi-Dimensional Array

1. Rules for one-dimensional array is applicable to two-dimensional array.
2. The total number of element in a multi-dimensional array must not exceed the product of its sizes.
3. One or more null array sizes is not allowed.

Syntax:

<data type> identifier[<size>][<size>] = { (<elements1>, <elements2>, …, <elementN>), (<elements1>, <elements2>, …, <elementN>)}.

**g. Expressions**

**1. Arithmetic**

An **arithmetic operator** in programming is a subset of these indicators or symbols that denote that a specific mathematical operation is needed.

Syntax:

<operand><arithmetic operator><operand>

Operands must be in the form of integer and float literals. Operands must also be in the form of identifiers. Arithmetic expression is also accepted as operand strings and character literals, and strings and character variable is not accepted as an operand. PEMDAS rule must be followed.

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| x + y | +xy |
| a \* b + c | bc- |
| b \* a / d | c+4\*b |

For Incrementation and Decrementation it must be in the following format:

<operand> <increment/decrement operator>

Or

<increment/decrement operator> <operand>

Operands must be in the form of integer literals. Operands must also be in the form of integer variables. Operands must not be either a character or a string. It can be placed anywhere in the expression.

Example:

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| x ++ x-- | x--- |
| ++x --x | ---x |
| 6++ 6-- | 6--- |
| ++6 6-- | ---6 |

1. **Boolean Expression**

A boolean expression is an expression that evaluates to a value of the Boolean data type: true or false. A boolean expressions can take several forms.

Rules for Boolean Declaration.

1. A statement should start with the data type, **boolean**, followed by an identifier then the boolean expression which results to TRUE or FALSE.
2. Comma is being used to separate multiple indexes.

Syntax:

<data type> identifier = <boolean expression>.

Comparison Operators such as ==, !=, <, >, <=, and >= produce boolean expressions by comparing the expression on the left side of the operator to the expression on the right of the operator and evaluating the result as True or False.

Example:

boolean sample = (1>5) then,

boolean sample = false.

**h. Statements**

Statement is an instruction written in a high-level language that commands the computer to perform a specified action.

**1. Simple Statements**

* 1. **Assignments –** an assignment statement sets and/or re-sets the value stored in the storage locations denoted by a variable name.
     + Expressiom could be in a form of identifier, literal, an arithmetic expression or a boolean expression.

Syntax:

identifier = <expression>

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **=** | assignment operator which assigns the value of its right operand to its left operand |
| += | Adds the two operands together, and then assign the result of the addition to the left operand. |
| -= | Subtracts the two operands together, and then assign the result of the addition to the left operand. |
| \*= | Multiply the two operands together, and then assign the result of the multiplication to the left operand. |
| /= | Divide the left operand by the right operand, and assign the result of the division to the left operand. |
| %= | Perform modular division on the two operands, and assign the result of the division to the left operand. |

Example:

pi = 3.1416.

* 1. **Input**

The absorb function returns a string value. Input of values must be in this format:

Syntax:

identifier = absorb().

Example:

string a = absorb().

* 1. **Output**

The cast function is an output statement that takes a String as its parameter. Expressions that return a non-string value must be enclosed with a parenthesis and converted to string or can be concatenated to it using the plus sign (+). Output of values must be in this format:

Syntax: cast(<statement>).

Example:

cast(“The value of pi is 3.14”).

Output: The value of pi is 3.14

cast(“The sum is” + (5+56)).

Output: The sum is 61

* 1. **Conditional Statements**

Conditional statement must be in the following format:

1. **if statement** – it must have at least one statement.

Syntax:

if (<condition>) {

<statement(s)>.

}

Example:

if (a < b) {

c = a + b.

}

1. **else-if statement** – it must at least one statement

syntax:

if (<condition>) {

<statement(s)>.

} else if (<condition>) {

<statement(s)>.

} else {

<statement(s)>.

}

Example:

if (year\_alice < year\_ben) {

cast(“Alice is younger than Ben ”.

} else if (year\_alice == year\_ben) {

cast(“Alice has the same age as Ben”).

} else {

cast(“Alice is older than Ben”).

}

1. **nested if-else**

if (<condition>) {

if (<condition>) {

<statement(s)>.

} else {

<statement(s)>.

}

} else {

<statement(s)>.

}

Example:

if (choice == ‘b’) {

if (answer > 4) {

cast(“You are a winner!”);

} else {

cast(“You are a loser!”).

}

}

1. **switch Statement**

It must have at least one statement.

Syntax:

switch (identifier) {

case <literal>:

<statement(s)>. break.

case <literal>:

<statement(s)>. break.

case <literal>:

<statement(s)>. break.

.

.

.

case <literal>:

<statement(s)>. break.

default: <statement(s)>

}

Example:

switch (varMood) {

case 0:

cast(“Happy”). break.

case 1:

cast(“sad”). break.

case 2:

cast(“Lonely”). break.

case 3:

cast(“Neutral”). break.

default: cast(“Invalid Input”).

}

1. **Looping**
   1. **for loop**

* must have at least one statement
* initialization must be in the following format:

identifier = <value>.

Syntax:

for(<initialization>. <condition> <inc/dec>) {

<statement(s)>.

}

Example:

ctr = 3.

for(n=0. n<ctr. ctr++){

cast(“Hello World!”).

}

* 1. **do while**
* must have at least one statement.

Syntax:

do {

<statement(s)>.

} while (<condition>)

Example:

ctr =3.

n=0.

do {

Cast(“World!”)

n++.

} while (n < ctr).

* 1. **while**
* must have at least one statement

Syntax:

while (<condition>) {

<statement(s)>.

}

Example:

ctr =3.

n=0.

while (n<ctr) {

cast(“World!”)

n++.

}