

**Pamantasan ng Lungsod ng Maynila**

*(University of the City of Manila)*

Intramuros, Manila

**College of Engineering and Technology**

**Computer Science Department**

**CSC 414-1 AUTOMATA AND LANGUAGE THEORY**

**Game Engine++ (GE++)**

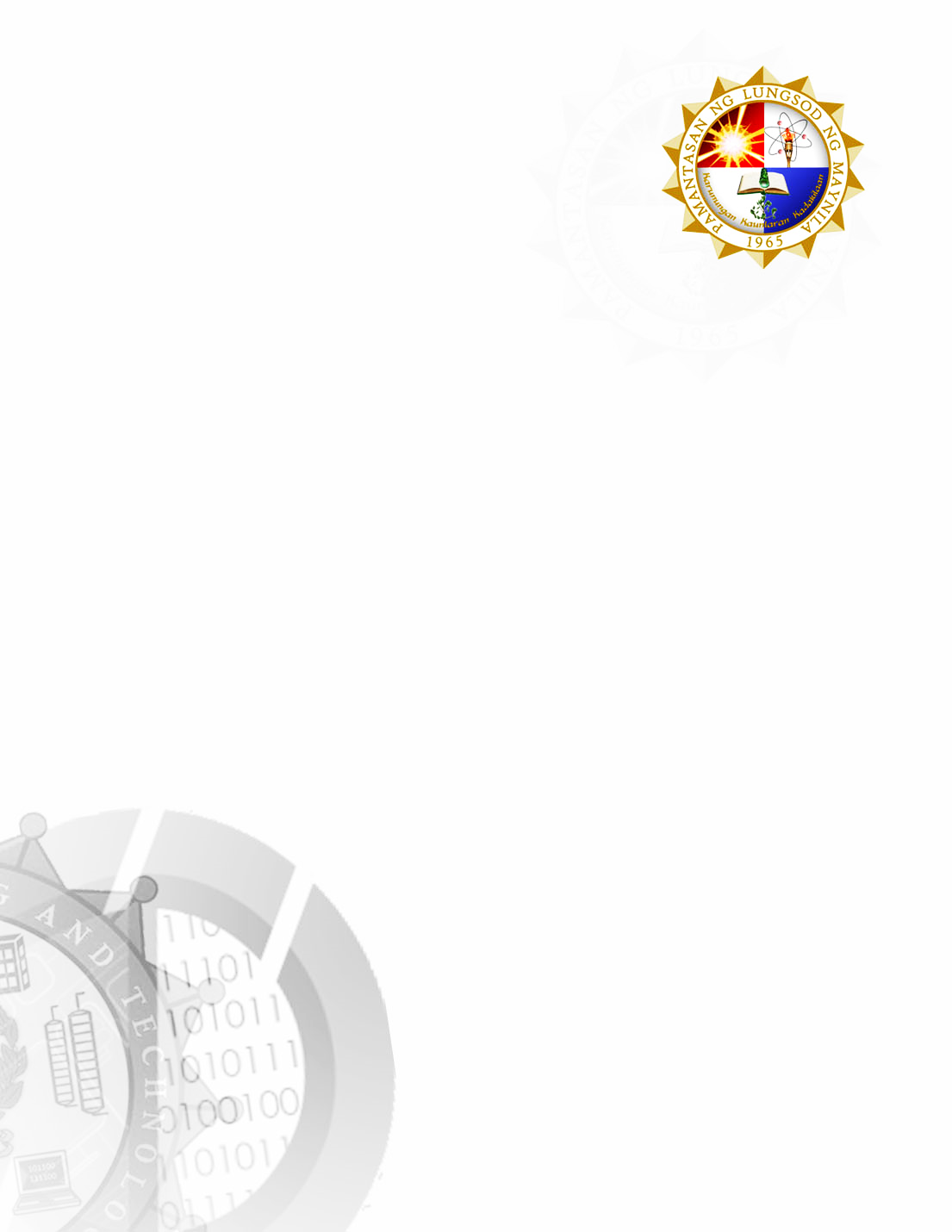
Submitted by:

Arañez, Katherine Mae A.

Guialil, Jamillah S.

Lanto, Dale Nardeen M.

Faustino, Bradley Jan E.

Rivera, Joeylene T.

Submitted to:

Prof. Leisyl M. Ocampo

Date:

July 24, 2017

**TABLE OF CONTENTS**

Title Page

Table of Contents

Language Overview

Program Structure

General Rules

Keywords

Reserve Words

Reserve Words for literal values

Reserve Functions

Reserve Symbols

Regular Definition

Regular Expression

Reserve Words

Reserve Words for literal values

Reserve Functions

Reserve Symbols

Identifiers

Literal

Other Delimiters/Separators

Comment

Transition Diagram

Specific Rules

Context Free Grammar

**I. Language Overview**

GE++ (Game Engine++) Compiler was built based on the structure of C++ and Java. The compiler was made using C#. The name of the compiler GE++ or Game Engine++ refers to the software framework designed for the creation and development of video game. It is also a reference to the increment geek pun used in naming the C++.

Some of the reserved words were derived from game concepts and terminologies. There are similarities in the symbols and other reserved words used to C++ and Java.

GE++ is a case sensitive programming language and was developed by five students from College of Engineering and Technology taking Bachelor of Science in Computer Studies major in Computer Science of Pamantasan ng Lungsod ng Maynila.

**II. Program Structure**

**Syntax:**

*loading engines.*

*global declarations.*

~~comments

*dataType* **map** *mapName*(){

~~comments

*local declarations.*

*statement(s).*

}

**state** **main()** {

~~comments

*local declarations.*

*statement(s).*

}

**III. General Rules**

1. GE++ is a case sensitive programming language.
2. Programs written in GE++ have the extension **.eng**.
3. A GE++ program starts with the loading of engines. **ENGINEs** are libraries you can import to the program. In loading an engine you use the syntax, load *engine*. Examples of engines are String\_Engine and Array\_Engine.
4. In creating a program, a **state main** function is created in which it is enclosed with a pair of **curly braces**. The **state main** function loads on default. Local declarations and statements are written inside the **state main** function. It is a must for every GE++ Program to have a **state main** function.
5. Global declarations and loading of engines are written before the other functions.
6. GE++ accepts functions called **map** which contains a block of statements and local declarations. There are two types of **map:**
   1. **Non-void map –** It is a map that returns a value. To indicate that it is a non-void map, the method should contain a return statement and have a non-void data type like int or float.
   2. **Void map** – It is a map that doesn’t return a value. The data type for void map is “void”
7. In calling a map, you must write the word *run* following the prototype, run *mapName*.
8. The limit is 100 parameters for every function definition. Each parameter consists of a data type, followed by an identifier. Each parameter must be separated with a comma (,).
9. The limit is 100 arguments for every function call. Each argument consists of a data type, followed by an identifier. Each parameter must be separated with a comma (,).
10. Non-void function must start with a non-void data type, like int or float, and followed by the word *map* then the name, *dataType* map mapName(). This is the header of the function and may contain a block of statements and local declarations.
11. Void function must start with void followed by the word *map* then the name, void map mapName().
12. Every function must be followed by a pair of curly braces.
13. Multiple declaration of a function with the same function name is not allowed.
14. A period(.) is used as a statement separator-terminator in creating a line statement.
15. Each line of code has a basic instruction. Ex. ctr = 0.
16. Identifiers must start with a letter followed by any alphanumeric characters. Minimum number of characters is 1, maximum is 20. Hyphen and underscore are the only special characters allowed.
17. The format of a single line comment is a message that starts with two curly dashes (~~). While the format of a block comment is enclosed with three curly dashes (~~~).
18. In reading expressions, a left to right operator precedence rule is followed aside from the PEMDAS rule.
19. The cast statement is used to display an output. The format is the statement enclosed with a pair of double quote inside a pair of parenthesis. Ex. cast(“The value of pi is 3.14”).
20. The absorb statement is used to receive an input. It returns a string value.
21. The limit is 100 characters for case labels for a switch statement.

**IV. Keywords**

**a. Reserve 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 |

**b. Reserve 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 |

**c. Standard Identifiers**

|  |  |
| --- | --- |
| **Standard Identifiers** | **Meaning** |
| absorb | Receives the input data, and it returns a string value. |
| cast | Prints out the the string |
| load | used at the beginning of the source file to import engines |

**d. Reserve Symbols**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **Assignment Operator** | |
| **=** | 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. |
| **Increment and Decrement Operators** | |
| **++ increment** | |
| ++a (Prefix) | adds 1 before the operand is evaluated |
| a++ (Postfix) | adds 1 after the operand is evaluated |
| **-- decrement** | |
| --a (Prefix) | subtracts 1 before the operand is evaluated |
| a-- (Postfix) | subtracts 1 after the operand is evaluated |
| **Arithmetic Operators** | |
| + | Addition (adds) |
| - | Subtraction (subtracts) |
| \* | Multiplication (multiply) |
| / | Division(divides) |
| % | Modulo (retains remainder |
| **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 |
| **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 |
| **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 |

**V. Regular Definition**

|  |  |
| --- | --- |
| Name | Definition |
| alpha | a-z, A-Z |
| num | 0-9 |
| num1 | 1-9 |
| alphanum | alpha, number |
| ascii | All printable characters |
| arithmetic | +, -, \*, /, % |
| logical operators | &, |, !, ~ |
| relational operators | <, >, = |
| separators | ), :, ., space |
| separators1 | (, ., space |
| Delim1 | num, Identifier, space, ( , ! |
| Delim2 | logical operators, separators |
| Delim3 | num, identifier, separators1 |
| Delim4 | Delim6, separators |
| Delim5 | “, identifier, num, space, ~, ! |
| Delim6 | logical operators, relational operators, arithmetic, ., space |
| Delim7 | Space, newline |
| Delim8 | space, ., =, , |
| Delim9 | +, separators, separators1 |
| Delim10 | space, {, newline |
| Delim11 | identifier, space, period |
| n | any positive integer |

**VI. Regular Expression**

**a. Reserve Words**

|  |  |  |
| --- | --- | --- |
| **Reserved Words** | **Regular Expression** | **Token** |
| boolean | (b)(o)(o)(l)(e)(a)(n) | boolean |
| break | (b)(r)(e)(a)(k) | break |
| byte | (b)(y)(t)(e) | byte |
| case | (c)(a)(s)(e) | case |
| char | (c)(h)(a)(r) | char |
| const | (c)(o)(n)(s)(t) | const |
| continue | (c)(o)(n)(t)(i)(n)(u)(e) | continue |
| default | (d)(e)(f)(a)(u)(l)(t) | default |
| do | (d)(o) | do |
| else | (e)(l)(s)(e) | else |
| float | (f)(l)(o)(a)(t) | float |
| for | (f)(o)(r) | for |
| goto | (g)(o)(t)(o) | goto |
| if | (i)(f) | if |
| int | (i)(n)(t) | int |
| return | (r)(e)(t)(u)(r)(n) | return |
| string | (s)(t)(r)(i)(n)(g) | string |
| switch | (s)(w)(i)(t)(c)(h) | switch |
| void | (v)(o)(i)(d) | void |
| while | (w)(h)(i)(l)(e) | while |

**b. Reserve Words for literal values**

|  |  |  |
| --- | --- | --- |
| **Reserved Words** | **Regular Expression** | **Token** |
| false | (f)(a)(l)(s)(e) | false |
| null | (n)(u)(l)(l) | null |
| true | (t)(r)(u)(e) | true |

**c. Standard Identifiers**

|  |  |  |
| --- | --- | --- |
| **Standard Identifiers** | **Regular Expression** | **Token** |
| absorb | (a)(b)(s)(o)(r)(b) | absorb |
| cast | (c)(a)(s)(t) | cast |
| load | (l)(o)(a)(d) | load |

**d. Reserve Symbols**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Regular Expression** | **Token** |
| **Assignment Operators** | | |
| **=** | (=) | **=** |
| += | (+)(=) | += |
| -= | (-)(=) | -= |
| \*= | (\*)(=) | \*= |
| /= | (/)(=) | /= |
| %= | (%)(=) | %= |
| **Increment and Decrement Operator**  **++ increment** | | |
| ++a (Prefix) | (+)(+) | ++a (Prefix) |
| a++ (Postfix) | (+)(+) | a++ (Postfix) |
| **-- decrement** | | |
| --a (Prefix) | (-)(-) | --a (Prefix) |
| a-- (Postfix) | (-)(-) | a-- (Postfix) |
| **Arithmetic Operators** | | |
| + | (+) | + |
| - | (-) | - |
| \* | (\*) | \* |
| / | (/) | / |
| % | (%) | % |
| **Relational Operators** | | |
| == | (=)(=) | == |
| != | (!)(=) | != |
| < | (<) | < |
| > | (>) | > |
| <= | (<)(=) | <= |
| >= | (>)(=) | >= |
| **Logical Operators** | | |
| && | (&)(&) | && |
| || | (|)(|) | || |
| ! | (!) | ! |
| **Bitwise Logical Operators** | | |
| & | (&) | & |
| ^ | (^) | ^ |
| | | (|) | | |
| ~ | (~) | ~ |

**d. Identifiers**

|  |  |  |
| --- | --- | --- |
| **Identifier, Literals, Constants** | **Regular Expression** | **Token** |
| Identifier | (alpha) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) (alphanum/\_/-/Ø) | Identifier |

**e. Literal**

|  |  |  |
| --- | --- | --- |
| **Literals** | **Regular Expression** | **Token** |
| Boolean Literal | (true / false) | Boolean Literal |
| Byte Literal | (num1) (num/Ø) (num/Ø) | Byte Literal |
| Character Literal | (‘) (ascii/Ø) (’) | Character Literal |
| Integer Literal | (-/Ø) (num1) (num/Ø)6 | Integer Literal |
| Float Literal | (-/Ø) (num) (num/Ø)5 (.) (num) (num/Ø)5 | Float Literal |
| String Literal | (“) (ascii/Ø)n (”) | String Literal |

**f. Other Delimiters/Separators**

|  |  |  |
| --- | --- | --- |
| **Other Delimiters/Separators** | **Regular Expression** | **Token** |
| parenthesis | ( ( / ) ) | parenthesis |
| semicolon | (:) | semicolon |
| period | (.) | period |
| braces | ({ / } ) | braces |
| brackets | ( [ / ] ) | brackets |

**g. Comment**

|  |  |  |
| --- | --- | --- |
| **Comment** | **Regular Expression** | **Token** |
| Single Line Comment | (~)(~)(ascii)n | Single Line Comment |
| Block Comment | (~)(~)(~)(ascii)n(~)(~)(~) | Block Comment |