# Language Reference Manual

## 1. Introduction

This manual describes the Logo++ language. The manual describes the proposed syntaxes and semantics of the language. Logo++ is an interpreted language designed to navigate the graphical user interface (GUI) in which a turtle is moving around and drawing lines. This language is designed to be an education tool serving students as an introduction to the world of programming. For language tutorial, please refer to the *Language Tutorial for Logo++*.

## 2. Lexical analysis

The Logo++ input is translated and interpreted in several stages. The input is stripped into a stream of tokens and analyzed by a parser.

### 2.1. Tokens

Tokens of Logo++ fall into one of the following five classes: identifiers, keywords, string literals, numeric constants, and operators. Tokens are stripped off from input and separated by white spaces. White spaces are blanks, tabs, newlines, and comments. They are ignored by the lexical analyzer except when they are used to separate tokens. Some white spaces are required to separate identifiers, keywords, and constants.

### 2.2. Comments

A comment either begins with a hash character (#) and terminates with another, or starts with two hash characters and ends with a newline character. Comments cannot be nested, and cannot be used within a string literals.

### 2.3. Identifiers

An identifier is a sequence of letters (A-Z | a-z) and digits (0-9). The first character must be a letter. Case is significant. Identifiers is limited to at most 31 characters.

### 2.4. Keywords

There are identifiers reserved as keywords, and cannot be used otherwise.

#### 2.4.1. Control flow

The following keywords are reserved for control flow:

```
if else for while Repeat break
```

#### 2.4.2. Commands

Commands are the core functions of Logo++. The following is the list of commands reserved as keywords:

```
Forward Back Left Right Clearscreen Home Wrap Fence

PenUp PenDown Teleport GPS ShowTurtle HideTurtle

FD BK LT RT CS ST HT
```

#### 2.4.3. Definition

The following keywords are reserved for assigning variables, defining functions, or built-in functions:

```
Function Set Print
```

### 2.5. Literals

String literal is a sequence of characters enclosed by double quotes (“). The backslash is used to introduce character with special meaning, such as newline and the quote character. Strings can be concatenated into a single string.

### 2.6. Numeric constants

All numeric constants used in Logo++ are treated as double. Although the language differentiates between integer and floating-point, there is no other numeric type specification.

### 2.7. Operators

The following tokens are operators:

```
+       -       *       /       ^       %

>       <       >=      <=      =       !=

&&      ||      !       ,
```

## 3. Meaning of identifiers

Identifiers refer to the names of functions, commands, or variables. Each identifier refers to the binding of that identifier established in the innermost block. A block is a module or function body. The scope of each identifier is static to the block. The whole interpreter is a global block.

### 3.1. Variables

There are 2 kinds of variables in Logo++. One denotes numerical values and the other represents string sequences.

#### 3.1.1. Numerical variables

Numerical variables are used to describe all numerical values and are regarded as double type. In Logo++, both integers and floating points are treated as numerical variables and have double type accuracy.

There are operators that can be used on numerical variables, which are the additive operator ‘+’, the subtraction operator ‘-’, the multiplicative operator ‘*’, the division operator ‘/’, and the power operator ‘^’. The use of these operators will be introduced in section 5.

Numerical variables can also represent the result of logical expressions, serving as Boolean variables. All non-zero numerical variables are treated as *true*, and zero-valued numerical variables are treated as *false*. This rule also applies to numerical constants.

#### 3.1.2. String variables

String variables are used to represent string literals. In Logo++, no type conversion between string variables and numerical variables is allowed. Logo++ provides a module containing basic functions to handle string variables.

#### 3.1.3. Use of variables

Variables must be initialized on the first time of use. Users do not need to and are not allowed to specify types of variables explicitly. Logo++ will automatically recognize them.

#### 3.1.4. Type conversion

Type conversions are handled automatically in Logo++. The result data type of binary operations follows the follow rule:

| | int | float |
|---|---|---|
| int | int | float |
| float | float | float |

### 3.2. Functions

Functions are user-defined procedures which take zero or more values as input and produce output by the last statement. Functions may or may not return a value depending on the last statement within the function block.

## 4. Commands

Commands are the reserved keywords that navigates the Logo++ graphic user interface (GUI).

### 4.1. Navigation commands

Navigation commands are the commands followed by one or more numeric constants or variables bound to numeric constants to navigate the movement of the turtle.

*Navigation commands:*

*Forward expression*

*Back expression*

*Left expression*

*Right expression*

*FD expression*

*BK expression*

*LT expression*

*RT expression*

*Teleport [expression, expression]*

Each of this commands takes a numeric type as argument. FD, BK, LT, and RT are shorthand of Forward, Back, Left, and Right respectively. Shorthand serves the same function as the corresponding command.

*Forward* moves the turtle forward, depending on where it is heading. The argument defines the travelling distance. The unit distance is 1 pixel on the screen.

*Back* moves the turtle backward. It has the same function as Forward except that the turtle is moving in the opposite (half a circle) direction.

*Left* turns the heading direction of the turtle to the left. The argument defines the angle (1/360 of a circle) it is turning. In default, the turtle is heading upward, perpendicular to the horizontal borders.

*Right* turns the heading direction of the turtle to the right. The argument is complimentary to Left.

*Teleport* directly set the position of the turtle. It takes a pair of bracketed arguments, each correspond to the x and y coordinates of the destination.

## 4.2. Non-navigation commands

Non-navigation commands are the commands that navigates the GUI as a whole.

*Non-navigation commands:*

*Clearscreen*

*Home*

*Fence*

*Wrap*

*PenUp*

*PenDown*

*CS*

*GPS*

*CS* is the shorthand of *Clearscreen*. It erases all the drawings on the screen and sends the turtle back to the origin, heading upward.

*Home* sends the turtle back to the origin, the center of the canvas, and also turns the turtle heading upward.

*Wrap* toggles off the fence on the border of the canvas. In wrap mode, the turtle can travels through the border to the opposite side, regarding the canvas like the map of the earth. In default, Logo++ is in wrap mode.

*Fence* toggles on the fence on the border of the canvas. In fence mode, the turtle is blocked on the boundary. It means once the turtle reaches the boundary, it will stop immediately. The default mode of the interpreter is wrap mode.

*PenDown* enables the turtle to actually draw on the canvas. In PenDown mode, once the turtle moves from one point to another, it will draw a line between these two points. In default, Logo++ is in PenDown mode.

*PenUp* disables the turtle to actually draw on the canvas. In PenUp mode, the turtle only moves without leaving trace on the canvas.

*GPS* returns the pairs of X and Y coordinates of the turtle's current position.

## 5. Expressions

The subsections retain the order of precedence of expressions. Left- or right- associativity is specified in each subsection for the operators.

### 5.1. Primary expressions

Primary expressions are variables, numeric constant, string literals, or expressions in parentheses.

*Primary-expressions:*

*Numeric constant*

*Variable*

*String*

*(expression)*

A numeric constant or a numerical variable is a primary expression. It has a double value.

A string literal or a string variable is also a primary expression. It is a sequence of characters.

A parenthesized expression is an identity to the primary expression inside the parenthesis. The parentheses affects only the precedence of the primary expression.

### 5.2. Function call

To call a function, directly use its name, and may need to have a list of arguments following it.

*Function-call:*

*Identifier(expressions$_{opt}$)*

### 5.3. Power

The power operator ^ is right-associative.

*Power-expression:*

*primary-expression* ^ *power-expression*

The operands of ^ must both have numeric types. The binary ^ operator denotes power.

### 5.4. Unary expressions

The unary operators -, ! are right-associative.

*Unary-expression:*

- *unary-experession*

*!unary-expression*

The operand of unary - operator must have numeric type. The result is the negation of its operand.

The operand of ! operator must have numeric type. The result is 1 if the operand is zero-valued, and 0 otherwise.

## 5.5. Multiplicative operators

The multiplicative operators *, /, % are left-associative.

*Multiplicative-expression:*

*Multiplicative-expression * unary-expression*

*Multiplicative-expression / unary-expression*

*Multiplicative-expression % unary-expression*

The operands of * and / must both have numeric types.

The binary * operator denotes multiplication.

The binary / operator denotes division. If both operands have integral type, the result would be rounded off to the nearest integer.

The binary % operator denotes remainder. Both operands must have integral type.

## 5.6. Additive operators

The additive operators + and – are left-associative.

*Additive-expression:*

*Multiplicative-expression*

*Additive-expression + multiplicative-expression*

*Additive-expression - multiplicative-expression*

The operands of + and – must both have numeric types.

The binary + operator denotes addition. (*and maybe concatenation)

The binary – operator denotes subtraction.

## 5.7. Relational operators

The relational operators <, >, <=, >= are left-associative.

*Relational-expression:*

*Additive-expression*

*Relational-expression < Additive-expression*

*Relational-expression > Additive-expression*

*Relational-expression <= Additive-expression*

*Relational-expression >= Additive-expression*

The operands of relational operators must have numeric types. These operators return 1 if the comparison relation is true and 0 otherwise. The type of this result is integer.

## 5.8. Equality operators

The equality operators =, != are left-associative.

*Equality-expression:*

*Relational-expression*

*Equality-expression = relational-expression*

*Equality-expression != relational-expression*

The operands of equality operators must have numeric types. Equality operator = returns 1 if both operands have same numeric value and 0 otherwise. Inequality operator != returns 1 if the operands have distinct numeric value and 0 otherwise. The type of this result is integer.

## 5.9. Logical AND operator

The logical AND operator && is left-associative.

*Logical-AND-expression:*

*Equality-expression*

*Logical-AND-expression && Equality-expression*

The operands of logical AND operator must have numeric types. It returns 1 if both operands are not zero-valued and 0 otherwise.

## 5.10. Logical OR operator

The logical OR operator || is left-associative.

*Logical-OR-expression:*

*Logical-AND-expression*

*Logical-OR-expression || Logical-AND-expression*

The operands of logical OR operator must have numeric types. It returns 1 if at least one of the operands is not zero-valued and 0 otherwise. The type of this result is integer.

## 5.11. Paired expressions

Paired expressions is presented by a pair of bracket with a comma separating two expressions. It is used for certain commands such as Teleport.

*Expression*

*Logical-OR-expression:*

*[Expression, Logical-OR-expression]*

### 5.12. Command expressions

In Logo++, most of the high-level expressions are command expressions.

*Command-expression:*

*Command*

*Command-with-expression*

*Command-with-expression*

*Command Expression*

All effects of command expressions are completed before the next command expression is executed.

## 6. Statement

Statements are executed in sequential order. Statements are executed for their effect, and do not have values.

*Statement:*

*Assignment-statement*

*Expression-statement*

*Conditional-statement*

*Iteration-statement*

### 6.1. Expression-statements

Expression statements are mostly variable assignment and function calls. Variable declaration is also a type of function call.

*Expression-statement:*

*expression$_{opt}$;*

### 6.2. Conditional statements

Conditional statements are if-else statements. Dangling else is associated to the closest if.

*Conditional statement:*

*if (expression) statement*

*if (expression) statement else statement*

The expression must have numeric type. If the expression does not equal to 0, the statement right after is executed. Otherwise, the second statement is executed only if the second form is present.

### 6.3. Iteration statements

Iteration statements specify looping.

*Iteration-statement:*

*Repeat expression [statement]*

*while (expression) statement*

*for identifier '=' expression:expression$_{opt}$:expression [statement]*

In the *Repeat* statement, the substatement is executed the number of expression times. Expression here must have an integer value.

In the *while* statement, the substatement is executed as long as the expression unequal to 0.

In the *for* statement, a variable is needed to be initialized as an index. The first expression is the initialization value. The second expression is the incremental value, by which the index will increase in value after each iteration. This expression is optional and is default as 1 when missing. The third expression is the termination value; the loop will perform its last iteration when the index equals to this expression.

## 7. Grammar

The following is a summary of the grammar that was defined throughout this manual. Notice that certain naming may be different:

```
primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | expression
        | command_expression
        ;

/* ----------------------- command expression ----------------------- */
command_expression_statement
        : command_expression
        ;

command_expression
        : command command_expression
        | command_with_expression command_expression
        |
        ;

command
        : GPS
        | origin
        | showturtle
        | hideturtle
        | wrap
        | fence
        | penup
        | pendown
        | clearscreen
        ;
```

```
command_with_expression
        : command_additive additive_expression
        | command_string STRING_LITERAL
        ;

command_additive
        : forward
        | back
        | left
        | right
        | setxy
        ;

command_string
        : print
        ;

forward
    :   c = 'FORWARD'
    |   c = 'FD'
    ;

back
    :   c = 'BACK'
    |   c = 'BK'
    ;

left
    :   c = 'LEFT'
    |   c = 'LT'
    ;

right
    :   c = 'RIGHT'
    |   c = 'RT'
    ;


setx
    :   c = 'SETX'
    ;

sety
    :   c = 'SETY'
    ;

setxy
    :   c = 'SETXY'
    |   c = 'TELEPORT'
    ;

getx
    :   c = 'GETX'
    ;

gety
    :   c = 'GETY'
    ;

getxy
    :   c = 'GETXY'
    |   c = 'GPS'
    ;

speed
    :   c = 'SPEED'
    ;

print
    :   c = 'PRINT'
    ;

clearscreen
    :   c = 'CLEARSCREEN'
    |   c = 'CS'
    ;

origin
    :   c = 'ORIGIN'
    |   c = 'HOME'
    ;

showturtle
    :   c = 'SHOW_TURTLE'
    |   c = 'ST'
    ;

hideturtle
    :   c = 'HIDE_TURTLE'
    |   c = 'HT'
    ;

wrap
    :   c = 'WRAP'
    ;

fence
    :   c = 'FENCE'
    ;

penup
    :   c = 'PEN_UP'
    ;

pendown
    :   c = 'PEN_DOWN'
    ;
```

```
/* ----------------------- additive expression ------------------------ */

unary_expression
        : primary_expression
        | unary_operator primary_expression
        ;

unary_operator
        : '-'
        | '!'
        ;

multiplicative_expression
        : unary_expression
        | multiplicative_expression '*' unary_expression
        | multiplicative_expression '/' unary_expression
        | multiplicative_expression '^' unary_expression
        ;

additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        ;

/* ----------------------- relational expression ---------------------- */

relational_expression
        : additive_expression
        | relational_expression '<' additive_expression
        | relational_expression '>' additive_expression
        | relational_expression '<=' additive_expression
        | relational_expression '>=' additive_expression
        ;

equality_expression
        : relational_expression
        | equality_expression '=' relational_expression
        | equality_expression '!= ' relational_expression
        ;

and_expression
        : equality_expression
        | and_expression '&&' equality_expression
        ;

or_expression
        : and_expression
        | or_expression '||' and_expression
        ;

expression
       : or_expression
```

```
         | expression ':' or_expression
         ;

paired_expression
       : '[' paired expression ',' expression ']'
         ;


assignment_expression
        : 'SET' IDENTIFIER expression


/* -----------------------  statement ------------------------ */

statement
        : assignment_statement
        | expression_statement
        | conditional_statement
        | iteration_statement
        | command_expression_statement
        ;

statement_list
        : statement
        | statement_list statement
        ;

expression_statement
        : expression
        ;

assignment_statement
        : assignment_expression
        ;

conditional_statement
        : IF '(' expression ')' statement_list
        | IF '(' expression ')' statement_list ELSE statement_list
        ;

iteration_statement
        : WHILE '(' expression ')' statement_list
        | FOR IDENTIFIER '=' expression statement_list
        | REPEAT IDENTIFIER '[' statement_list ']'
        ;
```