text++

Language Reference Manual (v. 1.0)

```
Joi Anderson - jna2123      // Manager + Tester
Klarizsa Padilla - ksp2127  // Language Guru + Tester
Maria Javier - mj2729       // System Architect + Tester
```

Contents

1. Introduction

text++ is a markup language designed for the production of technical documentation in an intuitive programming form. Unlike other templating languages like LaTeX, text++ is a markup language with algorithmic computing capabilities, allowing programmers to write documents as efficiently as they would write code.

2. Lexical Conventions

This section covers the text++ lexical convention for comments and tokens. There are six kinds of tokens: identifiers, keywords, constants, strings, operators, and separators. Blanks, tabs, newlines, and comments separate tokens, but they otherwise have no syntactic significance.

2.1 Comments

A single line comment in text++ starts with the // characters and extends to the end of the physical line of code. A single line comment can exist at the start of a line or following whitespace or code.

```
//  This is a comment
```

Multi-line comments start with the /* characters and terminate with the */ characters, ignoring all other characters encapsulated between the start and terminating characters.

```
/* This is a comment.
    It can have multiple lines. */
```

2.2 Identifiers

An identifier is a sequence of letters and digits, and the first character must be alphabetic. Identifiers must start with an alphabetic character, including the '_' character. Identifiers are case-sensitive.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise.

- `if`
- `else`
- `elif`
- `for`
- `while`
- `continue`
- `break`
- `return`
- `bool`
- `true`
- `false`
- `def`
- `dec`

2.3.1 Type-specifiers

Type-specifiers include bool, int, float, and string.

2.4 Constants

There are four types of constants, each with their own type, form and value :

2.4.1 Integer Constants

An integer literal is a sequence of digits, represented by characters [0-9]. Integer constants have type `int`.

2.4.2 Float Constants

A floating constant consists of an integer part, a decimal point, and a fraction part. Float constants have type `float`.

2.4.3 Character Constants

A character constant is a symbol enclosed in single quotes, such as 'x'.

Non printable characters can be represented via an escape sequence.

- Newline: '\n'
- Tab: '\t'
- Carriage return: '\r'

2.4.4 String Constants

Strings are marked with double quotes. The compiler will terminate strings with ‘\0’, the null byte, so that programs scanning a string can find its end.

2.6 Operators

An operator character signifies that an operation should be performed. The operators `[]`, `()`, and `{}` are used to encapsulate expressions and must occur in pairs.

Operator can be one of the following:
+ - * / % = == < <= > >= ! || mm (decrement) pp (increment)

2.6 Separators

A separator is a symbol between each element. Separator tokens include ‘,’ ‘;’ and whitespace is ignored. Separators are allowed in the following syntax:

Arrays:

```
Array a = [1,2,3,4,5]
A[0:3] = 9
```

3. Declarations

3.1 Function Declarations

Functions are declared as:

```
@ def functionName(type parameter){
      // local variables
      // statements
}
```

3.2 Variable Declarations

Variables are declared as:

```
 @ dec variableName[expression];
```

A variable may have its value updated, as long as its type remains consistent.

```
@ dec age[23];
@age                // 23
@age[@age + 3];
@age               // 26
@age['a'];        // Compiler error.
```

4. Expressions

Precedence of operators follows a standard order of operations (GEMDAS - Grouping symbols, Exponents, Multiplication, Division, Addition, Subtraction). text++is a left-associative language (evaluated left to right, after the application of order of operations).

4.1 Primary Expressions

4.1.1 Identifier

An identifier (like a variable) is a primary expression whose type is not required to be defined in its declaration.

4.1.2 Constant

Character, boolean, and integer.

4.1.3 String

A string is a primary expression.

4.2 Unary Operators

```
   -  expression
```

Can be applied to the int type. The result is the negative of the expression.

```
  ! expression
```

Logical negation operator. Applicable for type boolean.

```
expression pp
```

The left-value expression is incremented. Applicable to type int.

```
expression mm
```

The left-value expression is decremented. Applicable to type int.

4.3 Multiplicative Operators

```
expression * expression
```

The binary * operator indicates multiplication. Applicable to type int.

```
expression / expression
```

The binary / operator indicates division. Applicable to type int.

```
expression % expression
```

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int. The remainder keeps the sign of the dividend.

4.4 Additive Operators

```
expression + expression
```

The result is the sum of the expressions. Applicable to type int.

```
expression - expression
```

The result is the difference of the operands. Applicable to type int.

4.5 Relational Operators

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

The operators < , > , <= , and >= all yield false if the relation is false and true if the relation is true.

4.6 Equality Operators

```
expression == expression
expression != expression
```

The == and the != operators are analogous to the relational operators except for their lower precedence. Thus `a < b == c < d` is true whenever `a < b` and also `c < d`.

4.7 Boolean Operators

```
expression && expression
```

The && operator returns true if both its operands are true, false otherwise. The second operand is not evaluated if the first operand is false.

```
expression || expression
```

The || operator returns true if at least one of its operands is true, false otherwise.

5. Statements

Statements are executed in sequence.

5.0 End of Statement

The end of each statement is marked by a single '\n'.

5.1 Expression Statements

The majority of statements are expression statements, taking the form:

```
expression
```

These statements are usually assignments or function calls.

5.2 Conditional Statements

```
@if (expression) statement

@if(expression) {
   statement
}

@else {
   statement
}
```

If the expression is true, the (first) statement is executed. If the expression is false and there is an else, the second statement is executed. The elseless if problem is resolved by attaching an else to the last encountered if.

5.3 While Statements

```
@while(expression) {
   statement
}
```

The statement is executed as long as the expression is true. The evaluation of the expression occurs after each execution of the statement.

5.4 For Statements

```
@for(expr1; expr2;expr3) {
   statement
}
```

expr1 specifies initialization for the loop, expr2 is a test condition (evaluated before each iteration), and expr3 is an increment specification. The loop exits when expr2 is false.

5.5 Return statements

```
return
return (expression)
```

A function returns to its caller via a return statement. The second case returns the value of the expression. If the type expected by the caller does not match that of the return statement, an error will be thrown.

6. Interpolation

Built-in “math mode” or interpolation may be called using the @{} for multi line math blocks or @() for single line math blocks

```
return @(1+2)  //Returns 3
```

7. Scope Rules

7.1 Variable Scope

Variables declared outside of functions have global scope and can be accessed anywhere within the program. If declared within a function, variables only remain in scope for the duration of the function’s execution. Parameters passed into a function as arguments are declared as local variables within the scope of the function.

7.2 Function Scope

A function may not be called before it has been declared. All functions have global scope by default.

8. Arrays

8.1 1D Array

A 1D array is declared by:

```
@ myarray[[1,2,3,4]];
```

where myarray is the name of the array, and the type is int.

9. Separators

A separator is a symbol between each element. Separator tokens include ‘,’ ‘;’ and whitespace is ignored. Separators are allowed in the following syntax:

```
@ myarray[[1,2,3,4]];
```

10. Styling Operators

10.1 Header Operations

```
@mystring()
{ return "string is" @h6 ("this")}
```

There are 6 format types for headers. To call a particular type of header styling on strings one must use @h followed by the particular number, between 1 and 6, associated with the formatting the author desires.

10.2 Emphasis Operations

`@b(@mystring)`	bolds the value of mystring
`@i(@mystring)`	italicizes the value of mystring
`@u(@mystring)`	underlines the value of mystring

11. Formatted Output

The user can export their code in a formatted PDF using

```
render();
```

The PDF extension of text++ will interpolate values before formatting the text into the final PDF document.