# blur reference manual

Dexter Callender | dec2148
Tim Goodwin | tlg2132
Daniel Hong | sh3266
Melissa Kaufman-Gomez| mhk2149

# 1. Introduction

Blur draws inspiration from traditional ASCII art and pixel manipulation. Blur is a lightweight programming language that focuses on the manipulation and presentation of data in Euclidean, matrix-like representations. It provides the building blocks to allow the programmer to edit this ASCII art, for example, use only certain characters, limit the number of different characters used, control the density, etc.

# 2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and separators. Blanks, tabs, newlines, and comments separate tokens, but they otherwise have no syntactic significance.

## 2.1 Comments

/* this is a comment */

## 2.2 Identifiers (like variables)

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.

- char
- boolean
- int
- string
- null
- for

- while
- if
- else
- void
- return
- true
- false

2.3.1: Type-specifiers
Type-specifiers include char, boolean, int, string.

## 2.4 Constants

There are three types of constants in Blur:
2.4.1 Integer Constants
An integer literal is a sequence of digits, represented by characters [0-9].
2.4.2 Character Constants
A character constant is a symbol enclosed in  single quotes. Non-printable characters can be represented via an escape sequence.

- Newline --- \n
- Tab ---- \t
- The single quote --- \'

2.4.3 Strings
Strings are treated as arrays of characters, and 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.

# 3. Declarations

## 3.1 Function Declarations

Functions are declared as:

```
func void recurse (int x) {
      if (x == 0) {
            return 1;
      }
      else {
            return 1 + recurse(x);
      }
}
```

## 3.2 Variable Declarations

Variables are declared as:

<type> <variable_name> = *expression*;
A variable may have its value updated, as long as its type remains consistent.

```
int i = 5;
i = i + 3; // Results in 8.
i = ‘a’; // Compiler error.
```

## 3.3 Function Structure

```
func <return type> <function name> (<args and types>) {
      <local vars>
      <function body>
      <return>
}
```

# 4. Expressions

Precedence of operators follows a standard order of operations (GEMDAS - Grouping symbols, Exponents, Multiplication, Division, Addition, Subtraction). blur 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 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

4.2.1 *- expression*
Can be applied to the int type. The result is the negative of the expression.
4.2.2 *! expressions*
Logical negation operator. Applicable for type boolean.
4.2.3 *expression++*
The left-value expression is incremented. Applicable to type int.
4.2.4 *expression--*
The left-value expression is decremented. Applicable to type int.

## 4.3 Multiplicative Operators

4.3.1 *expression * expression*
The binary * operator indicates multiplication. Applicable to type int.
4.3.2 *expression / expression*
The binary / operator indicates division. Applicable to type int.
4.3.3 *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

4.4.1 *expression + expression*
The result is the sum of the expressions. Applicable to type int.
4.4.2 *expression − expression*
The result is the difference of the operands. Applicable to type int.

## 4.5 Relational Operators

4.5.1 *expression < expression (less than)*
4.5.2 *expression > expression (greater than)*
4.5.3 *expression <= expression (less than or equal to)*
4.5.4 *expression >= expression (greater than or equal to)*
The operators < , > , <= , and >= all yield false if the relation is false and true if the relation is true.

## 4.6 Equality Operators

4.6.1 *expression == expression (equal to)*
4.6.2 *expression != expression (not equal to)*
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 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.

## 4.8 expression || expression

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

## 4.9 expression ? expression : expression

The first expression is evaluated and if it is true, the result is the value of the second expression, otherwise the result is the value of the third expression. The types of the second and third should be the same.

## 4.10 Assignment operators

The left value is an identifier with a type. The stored value is on the righthand side, and is stored after the assignment operation.

4.10.1 *<type><identifier> = expression*

The value of the expression is of type <type>, and is stored in the identifier.

# 5. Statements

Statements are executed in sequence.

## 5.0 End of Statement

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

## 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 Compound Statements

Compound statements allow for multiple statements where one is expected.

```
compound statement:
      {list-of-statements}
```

## 5.3 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.

## 5.4 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.

## 5.5 For Statements

```
for (expression1; expression2; expression3) {
      statement;
}
```

expression1 specifies initialization for the loop, expression2 is a test condition (evaluated before each iteration), and expression3 is an increment specification. The loop exits when expression2 is false.

## 5.6 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. Functions

Built-in or predefined functions may be called using the function name,  and argument(s), if any.

```
foo(data);
```

Functions are declared and defined as described in **3.1**.

# 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 `int a[] = Array.build[i];` where i is the number of array elements, and the type is int.

## 8.2 2D Array

A 2D array is declared by `int a[][] = Array.build[i][j];` where the dimensions of the array are i x j.

# 9. Canvases

A canvas is a 2D array onto which the user can place points and “print” his/her ASCII art.

```
Canvas  c = { 10, 10, '*' }; // Creates a 10 x 10 canvas filled with '*'
c.put(2, 3, '+'); // Puts a '+' at coordinate (2,3)
```

## 9.1 Loading an image

Loading an image loads an array of pixel brightnesses.

```
Load(c, "ascii_image.jpg");
```

## 9.2 Saving to a file

To save a canvas to a file:

```
paint(c, "filename.text");
```

# 10. 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

Function Arguments:

foo(int a, int b)

# 11. Formatted Output

The user can output information for the purposes of debugging and string printing.
For printing to stdout without newline character:

```
Print();
```

For printing to stdout with newline character:

```
Println();
```

Formatted output of a dithered ASCII art image can be viewed in a text file.