Blox

Language Reference Manual

v1

October 26, 2016
Programming Languages and Translators
Stephen A. Edwards

Team Members

Name	UNI	Role
Paul Czopowik	pc2550	Manager
Naeem Bhatti	bnb2115	Language Guru
Tyrone Wilkinson	trw2119	System Architect
Jonathan Voss	jcv2130	Tester

Table of Contents

Introduction

This is a reference manual for the Blox programming language which documents the current standards.

Blox enables the programmer to solve a structural problem; given a set of resources and a mapping of constraints on those resources, it outputs the set of all possible solutions that conform to those mappings in the **Additive Manufacturing File** format (AMF) which is used for 3D printing.

The underlying problem faced in such a situation is how to:

1. Give programmers the freedom to actualize any object they desire.
2. Ensure the compiler has the capacity to interpret and manipulate such objects.

Blox resolves this dilemma by introducing the fundamental concepts around which the language is built: the Structure,Frame and Block.

Lexical Elements

This chapter describes the lexical elements that make up Blox source code after preprocessing. These elements are called tokens. There are five types of tokens: *identifiers*, *keywords*, *constants*, *operators*, and *separators*. White space, sometimes required to separate tokens, is also described in this chapter.

Identifiers

Identifiers are sequences of characters used for naming variables, functions, new data types, and preprocessor macros. You can include letters, decimal digits, and the underscore character '_' in identifiers. Lowercase letters and uppercase letters are distinct, such that foo and FOO are two different identifiers.

Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose. Here is a list of keywords recognized in Blox:

```
if  else  for  while  do  break  switch  case  default  return  void  NULL
int  float  char  bool
Create  Build  Rule  Join  Detach  Frame
Red  Orange  Yellow  Green  Blue  Violet  Indigo
```

Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type; you must use type casting to explicitly specify the type of a constant. There are integer constants, real number constants, character constants, and string constants.

Integer Constants

An integer constant is a sequence of digits assumed to be in base 10, so no prefixes are used.

```
0123
-45
9
```

Real Number Constants

A real number constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted, but not both. The exponent can be either positive or negative. Real number constants cannot be followed by e or E and an integer exponent.

```
4.2
.5
0.88
```

String Constants

A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is of type "array of characters". All string constants contain a null termination character (\0) as their last character. Strings are stored as arrays of characters, with no inherent size attribute. The null termination character allows string-processing functions know where the string ends.

Operators

Blox supports three types of operators.

1. Assignment operators
2. Comparison operators
3. Logical operators.

Assignment operators store values in variables. The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value.

Comparison operators are used to determine how two Frames relate to each other: are they structurally equivalent, are they composed of the same number of blocks, does one have more blocks than the other, does one have less blocks than the other, and so one, with the remainder only dealing with the number of blocks a Frame is composed of as well. When you use any of the comparison operators, the result is of type bool either true or false. The equal-to operator `==` tests two Frames for structural equality. The result is true if the Frames are equal, and false if the Frames are not equal.

```
if (reddy1 == reddy2)
      print("reddy1 is equal to reddy2");
else
      print("reddy1 is not equal to reddy2");
```

The not-equal-to operator `!=` tests two Frames for structural inequality. The result is true if the Frames are not equal, and false if the Frames are equal.

```
if (reddy1 != reddy2)
      print("reddy1 is not equal to reddy2");
else
      print("reddy1 is equal to reddy2");
```

The dot-equal-to operator `.=` only determines whether or not two Frames have the same number of Blocks. The result is of type bool, true if the Frames have equal number of Blocks, and false if the Frames have an unequal number of Blocks.

```
if (reddy1 .= reddy2)
      print("reddy1 is superficially equal to reddy2");
else
      print("reddy1 is not superficially equal to reddy2");
```

Beyond equality and inequality and block-equality, there are operators you can use to test if one Frame has less blocks, more blocks, blocks less-than-or-equal-to, or blocks greater-than-or-equal-to another Frame.

```
if (reddy1 < reddy2)
      print("reddy1 has less blocks than than reddy2");
if (reddy1 <= reddy2)
      print("reddy1 has blocks less than or equal to reddy2");
if (reddy1 > reddy2)
      print("reddy1 has more blocks than reddy2");
if (reddy1 >= reddy2)
      print("reddy1 has blocks greater than or equal to reddy2");
```

Logical operators test the truth value of a pair of operands. All non-zero expressions are considered true, while any expression evaluating to zero is considered false.

Operator	**Name**	**Example**
=	Assign	x = 6; /* The value of variable x is now 6 */
+	Addition	y = x + 4; /* The value of y is now 10 */
-	Subtraction	z = y - x; /* The value of z is now 4 */
*	Multiplication	a = x * y; /* The value of a is now 24 */
/	Division	b = y / 5; /* The value of b is now 2 */
^	Exponentiation	c = z ^ b; /* The value of c is now 16 */
%	Modulo	d = c % x; /* The value of d is now 4 */
++	Increment	e = c++; /* The value of e is 16 and c is now 17 */
--	Decrement	f = c--; /* The value of f is 17 and c is now 16 */
&&	Logical AND	expr1 && expr2 /* Returns true only if both expr are true, otherwise it returns false */
||	Logical OR	expr1 || expr2 /* Returns true if one or both expr are true, if both are false it returns false */
!	Logical NOT	!expr1 /* Returns true if expr1 is false, returns false if expr1 is true */
==	Equal To	z == d /* Returns true */ x == y /* Returns false */
!=	Not Equal To	z != d /* Returns false */ x != y /* Returns true */
>	Greater Than	x > y /* Returns false */ a > b /* Returns true */
>=	Greater Than or Equal To	a >= b /* Returns true */ z >= d /* Returns true */
<	Less Than	x < y /* Returns true */ a < b /* Returns false */
<=	Less Than or Equal To	x <= y /* Returns true */ z <= d /* Returns true */
[ ]	Access Array Element	array[0] /* Returns first element in array */
.	Access Object	object.mem /* Returns member mem in object */

	Member	

Precedence of Operators

```
[ ]    .
++    --    !
*    /    %    ^
+    -
>    >=    <    <=
==    !=
&&
||
=
```

Separators

A separator separates tokens. White space (see subsection) is a separator, but it is not a token. The other separators are all single-character tokens themselves:

```
(  )  <  >  ,  ;
```

White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character and the horizontal tab character. White space is ignored, and is therefore optional, except when it is used to separate tokens.

Comments

The characters **/*** introduce a comment, which is terminated with the characters ***/**. Block comments and nested comments are supported.

Data Types

Primitive Types

```
int  float  char  bool
```

Primitive types are predefined in the Blox language as the most basic data types available, and are named by their keywords. The numeric type include int, float, and char and boolean type holding either true or false.

```
int       from -2147483648 to 2147483647, inclusive
float     from 3.402,823,5 E+38 to 1.4 E-45, inclusive
char      from 0 to 65535
```

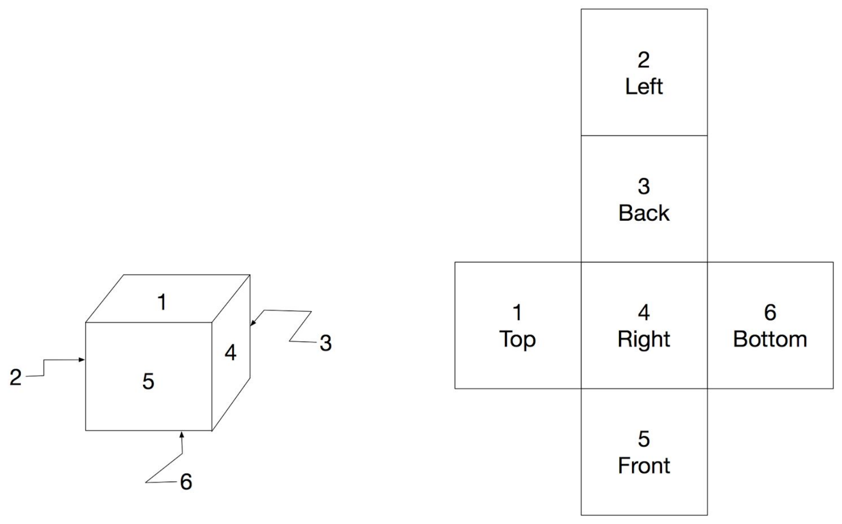
`bool`	**`true`** `or` **`false`**

Language Specific Types

The language has three hierarchical levels of abstraction and at each layer is an object that composes its “parent”. Structure is an object that contains 0 or more Frames and Frames are objects that contain 1 or more Block. A Structure is created and Frames can be joined to this Structure, constructing or building an element, thus it keeps track of the Frames and how they are joined. A Frame is a set of adjacent Blocks and keeps track of which face of each Block is available for a join.

Block

A Block is the fundamental abstract object of the language which serves the construction of Frames. A conceptual illustration is provided below.



A Block is a six element bit-vector and is aware only of which sides it has available (open) for joining and which sides are already joined or prohibited from joining by rules (closed). To accomplish this, a Block is stored as an boolean array of 6 elements where TRUE represents open and FALSE closed. The array is also offset by one so it is referred to in the range of 1-6 rather than the standard 0-5. Blocks are the only object to hold an orientation field: T, Bo, L, R, F, Ba signifying top, bottom, left, right, front, and back, respectively. Orientation is specified when performing Join operations.

Frame

A Frame is a data structure composed of a set of *n* blocks, were *n > 0* and it is specified at declaration. Frames are characterized by three spatial dimensions: *x, y,* and *z*. Additionally, the sRGB standard which utilizes the ITU-R BT.709 primaries, or one of seven built-in color keywords can be used to specify color.

Declaring and Initializing Frames

You declare a Frame and initialize its contents by using the Frame keyword, specifying its dimensions and its color, and then specifying its name. Declaration and initialization must be done together. Given below are two examples that both declare and initialize equivalent, red, 2 x 2 Frames, the latter example using optional values from the sRGB standard.

```
Frame<2,2,1,Red> reddy1;
Frame<2,2,1,1,0,0> reddy2;
```

You don't always have to declare and initialize a Frame before the main body of a program in order to use it. See the example program at the end of the manual. The syntax used to properly declare and initialize, or create, Frames must be one of the following:

```
Frame<dimension x,dimension y,dimension z,r value,g value,b value>
Frame<dimension x,dimension y,dimension z,color>
```

Structure

A Structure is an object that contains a set of attached Frames and a mapping of Join relationships between the Frames. It also contains a three-dimensional array that represents all the blocks that are in the Structure. Join operations are allowed by joining a Frame to a Structure. Initially, the first Frame is joined into an empty Structure. The Structure is an object that contains all the Frames joined to it and information on how they are joined. A conceptual illustration is provided below.

```
Structure Table
{
        Frames
        {
                Frame top<3,3,1>;
                Frame leg1<1,1,3>;
                Frame leg2<1,1,3>;
                Frame leg3<1,1,3>;
        }

        Joins
        {
                top(1,1,1, b) with leg1(1,1,3, t);
                top(1,3,1, b) with leg1(1,1,3, t);
                top(3,1,1, b) with leg1(1,1,3, t);
        }

        BlockArray
        {
                Block blockArray[3][3][3]
        }
}
```

Array

Arrays are declared by specifying the data type for its elements, its name, and then the number of elements it can store enclosed within a set of brackets. Array sizes are constant and indices start at value 0. Arrays can be initialized during or after declaration by referencing the index.

Syntax

```
type arrayName [arraySize];
type arrayName [arraySize] = {value1, value2...};
```

Example

```
int x[2];
int size[5] = {1, 10, 4, 5, 0};
size[0] = 10;
```

Set

Set supports add and remove functions.

Syntax

Set<data-type> identifier;	Declaration
identifier.add(identifier);	Adding an element
identifier.remove(identifier);	Removing an element

Example

```
Set<Frame> resources;
resources.add(frame1);
resources.add(frame2);
resources.remove(frame2);
```

Expressions

An expression consists of at least one operand and can include separators and operators. When an expression has subexpressions, the innermost expressions are evaluated first.

Example

```
( x + ( ( 5 * 74 ) / 37 ) - 4 )
```

In the above example, 5 * 74 evaluates to 370. Then, 37 is divided from 370, resulting in 10. 4 is subtracted from 10, resulting in 6, which is finally added to x. The outermost parentheses are not required.

Scope

Blox is a block-structured language, meaning the lexical scope of variables do not extend beyond the pair of curly braces in which they are declared.

Example

```
      {
            x = 5;
            {
                  x = 1;
                  y = 2;
            }
            print(x);   /* the value of x is 5 */
      }
```

Statements

All statements must end with the semicolon `;` character.

Grammar

```
statement:
      block-statement
      loop-statement
      jump-statement
      conditional-statement
```

Block statements

Syntax

{statement1; statement2; statement3; … }

Example

```
      {
            x = 1;
            y = 2;
      }
```

Conditional statements

Syntax

if (expression) {statements}
if (expression) {statements} **else** {statements}
if (expression) {statements} **else if** (expression) {statements} **else** {statements}

Example

```
if (x == y) {return x+1;}
if (x == y) {return x+1;} else {return x-1;}
if (x == y) {return x+1;} else if (x < 1) {return x-1;} else {return 0;}
```

Loop statements

For loop

Syntax

for (expression; expression; expression) { statements }

Example

```
int i;
for (i = 0; i < 10; i++)
{
      print("Hello ");
      print("World!");
}
```

While loop

Syntax

while (expression) { statements }

Example

```
int x = 0;
while (x < 10)
{
      print("Hello ");
      print("World!");
      x++;
}
```

Jump statements

Syntax

```
switch (expression)
{
      case constant-expression : statement
      case constant-expression : statement
      ...
      default : statement
}
```

Example

```
switch (result)
{
      case true:
             x = x + 1;
             break;
      case false:
             x = x - 1;
             break;
      default:
            break;
      return x;
}
```

Functions

Declaration

Syntax

Return-type identifier (parameter-list) {}
Parameter list = identifier parameter-list

Example

```
int add(int x, int y)
{
      int a = x + y;
      return a;
}
```

Calling

A function returns one data type or void. The function can be used as an expression itself.

Syntax

Return-type Function-name (parameter-list);

Example

```
removeUser(“Paul”, 100);          /* unused return statement */
User Paul = removeByID(12345);   /* assign returned user object to Paul */
if (pingIsOK(“192.168.1.1”)) {return true}; /* evaluated as a condition */
```

Built-in Functions and Keywords

Join

Join is a built in function which allows for connecting of Frames to produce a native Structure object. However, Frames may only be joined to Structures. Joining additional Frames to an existing Structure adds the Frame to the collection of Frames within that Structure. This also creates a join relationship entry.

Syntax:

    Join (Structure-identifier, Frame-identifier<x,y,z>);
    Join (Structure-identifier<x,y,z>, Frame-identifier<x,y,z>);

Example:

```
Create Structure A;
Create Frame<1,1,1> X;
Create Frame<1,3,1> Y;
Join(A, X<1,1,1>);          /* A was empty, now has one frame */
Join(A<1,1,1>, Y<1,3,1,U>); /* A now joins X and Y, both becoming A */
```

Build

The built-in function Build takes two parameters: Set<Frame> and Set<Rule> and generates the set of all possible objects that can be constructed using the set of Frames, abiding by the mapping of Rules to those Frames. If a Map value of **NULL** is passed, Build will abide by no rules except

```
Build(resources, rules);
```

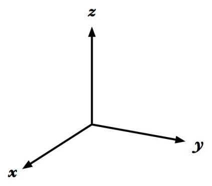
Create

The keyword Create dynamically allocates heap memory for a Frame or Structure only. Create takes a 3D position vector and a name. Optionally, a color can be specified. Create Frame<x,y,z> Frame dynamically allocates heap memory for n = x * y * z grid of Blocks. For example, Create Frame<3,1,1> will allocate memory for 3 * 3 * 1 = 9 Blocks.

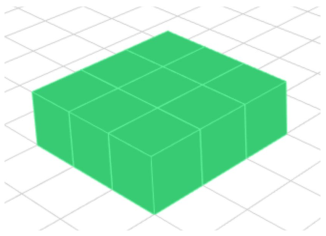
```
Create Frame<3,3,1,green> base;
Create Frame<10,100,4> X;
```

Sample Program

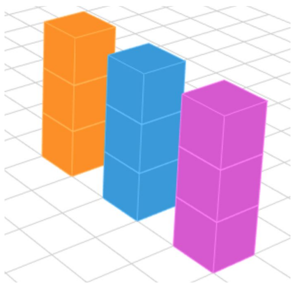
```
void main()
{
      /* make 9-block grid */
      Frame<3,3,1,green> base;

      /* make resources */
      Set<Frame> resources;
      for (int i = 0; i < 3; i++)
            resources.add(Frame<1,1,3>);

      /* add the base to the set of resources */
      resources.add(base);

      /* make rules for some/all resources */
      Set<Rule> rules;

      Rule rule1 = Join(base<1,3,1,U>, resources[0]<1,1,1,D>);
      rules.add(rule1);

      /*
       * using a set of given resources, build the set of objects possible that
       * conform to rules
       */
      Build(resources, rules);

      /* display output in AMF */
      Display;
}
```



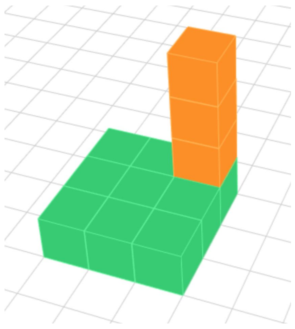
base:



res:



Rule1:



Build (showing two possible results):