

Boboscript

Boris Martin

February 15, 2017

Contents

1	Introduction	2
1.1	Purpose and main characteristics	2
2	Structure of a program	3
2.1	File layout	3
2.2	Function definitions	4
2.3	Expressions and statements	4
3	Type System	5
3.1	Primitive types and compound types	5
3.2	Classes	5

1 Introduction

The main goal of this document is to provide a formal, exact description of the programming language I'm working on, known as *Boboscript*. It will give indications about how to scan, parse, run or compile it, in a way that'll make the implementation almost obvious.

1.1 Purpose and main characteristics

Paradigm and data model Boboscript is meant to be a procedural, *C-like* language, which heavily encourage decoupling of *code* and *data*. Code is mainly represented through functions and data via POD structures. It supports primitive object-oriented programming, with garbage-collected objects that are the only way to have recursive data structures. No manual memory is allowed, excepted in low-level code (native C calls) : data is either stack-allocated or garbage-collected object.

Type system Boboscript is *statically* and *strongly typed* language. It is designed for compilation to C, and thus, use the C memory model.

Limitations Multi-thread support is not required, but could probably used through C native calls. Operation atomicity is not guaranteed in any case.

Modularity Every Boboscript file describes exactly one module, whose name must be declared in the beginning, with uppercase letters. It can be compiled to a binary format, with extension ".bobj". A full program consists of linked objects file, including one defining a module MAIN, which must contain the `main()` function.¹

All functions defined in a module are exclusive to this module, unless it is stated (in module declaration) that they must be exported.

¹In future versions, it could become possible to compile to a C library, with auto-generated headers.

2 Structure of a program

2.1 File layout

Module declaration Every Boboscript file must start with a Module declaration. It contains the name of the module, and between brackets, the signature of all functions that must be exported and the name of classes, structs and enums to export. It is a statement, which implies it **must** end with a semicolon.

```
Module MODULE_NAME {  
    Void arglessFunction(),  
    Int intToIntFunction(Int, Int),  
    NameOfAClassToExport,  
    MyEnum,  
    VectorStructure  
    ...  
};
```

Functions, types, constants and global variables declaration Once the module is declared, the remaining code consists of definitions. Types, functions, constants and global variables. All definitions are statements and must end with a semicolon. A typical file looks like this sample. ²

```
struct Vector2D {  
    Double x, y;  
};  
  
Double pi = 3.14;  
  
enum Op {PLUS, MINUS, TIMES, DIV};  
  
class LinkedListIntegerList {  
    LinkedListIntegerList(Int h, LinkedListIntegerList tail = null) {  
        _head = h;  
        _tail = tail;  
    };  
    Int _head;  
    LinkedListIntegerList _tail;  
};
```

²Class behavior will be defined *in extenso* later on.

2.2 Function definitions

Declaration A function definition starts with the *function* keyword, followed by the function name (starting with a lowercase letter or an underscore). Then comes the (potentially empty) arg list, between parentheses, in the form "Type name" and separated by commas. It is followed by an arrow (*->*) and the return type. Optionnally, a name can be added for documentation purposes, telling what the returned value represents.

Definition Definition comes right after declaration, is enclosed by brackets { } and ends with a semicolon. It contains statements and ends when it reaches a *return* statement, or the end of the block. Reaching end of a function without returning results in undefined behavior if the returned value is used.

Example

```
function abs(Double x) -> Double result {  
    return x > 0 ? x : -x;  
};
```

Inner functions A function can be defined inside a function, or, actually, inside any block of code. Its scope will vary accordingly.

2.3 Expressions and statements

Expression represent that can be evaluated as a value or an object reference. All variables are expressions, as well as function calls (including operators). Expressions are always used inside statements, or can be made into statement by appending a semicolon to them.

Statements are instructions. All type, variable and function definition are statements. Other statements include control flow, *return* etc. All statements end with a semicolon.

3 Type System

3.1 Primitive types and compound types

Primitive types are equivalent to their C counterparts. They are Int, Float, Double, Char, Bool.³ There is also the VOID_PTR special type, equivalent to *void** in C. It is designed to allow interfacing with C libraries (especially to implement the Bobo standard library).

Structures (*struct*) represent POD⁴ aggregates. Like their C counterpart, they can contain structured data, but not Object references neither instance of themselves. Namely, structures can contain other structures⁵, primitive types, fixed-size arrays and enums.

Enums (*enum*) work like in C : they are a bunch of constants converted to integers. But, unlike in C, they don't pollute global namespace : enum values must be accessed from scope resolution operator.

Fixed-size arrays are similar to the ones in C, but the array qualifier is in the type declaration. An array is typically defined like this :

```
Int[3] primes = {2,3,5};
```

Arrays can only be constructed from other arrays at initialization. Later on, it can only modified item by item. Construcitng an array from one of another size is **undefined behaviour**. An array does *not* know its own size.

3.2 Classes

Classes are similar to *structs*, but they can contain reference to objects⁶, including objects of its own class.

Methods A class can contain its own functions, called *methods*. They always have an implicit reference to the caller object, named **self**.

³Remember that all typenames must start with an uppercase letter.

⁴Plain old data

⁵Without recursion : if A has elements of type B, B can't contain As.

⁶"Object" refers to any instance of any class.

Access specifiers All members of a class (attributes and methods) can be *public*, *protected* or *private*. Public members can be accessed from anywhere, while private or protected only from inside. Protected members can also be used in derived classes (see later).