Boboscript

Boris Martin

February 16, 2017

Contents

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.

[1] In 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 globals 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 LinkedIntegerList {
    LinkedIntegerList(Int h, LinkedIntegerList tail = null) {
      _head = h;
      _tail = tail;
      };
    Int _head;
    LinkedIntegerList _tail;
};
```

[2] Class 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 POD[4] aggregates. Like their C counterpart, they can contain structured data, but not Object references neither instance of themselves. Namely, structures can contain other structures[5], 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 tipically 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 objects[6], 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**.

[3] Remember that all typenames must start with an uppercase letter.
[4] Plain old data
[5] Without 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).

Creation and lifetime An object must be created with the *new* operator. Objects are managed : they are destroyed when they aren't referenced anywhere.

```
MyClass object = new MyClass(constructorArg);
```

Inheritance Classes can inherit each other. Every class can have only one parent class, and all methods are virtuals. If B derives from A and C from B, a reference to A may refer to a A, a B or a C.

3.3 Type constructors

Type constructors construct types from other types. There are 3 type constructors.

Const A *const T* is a T that can't be modified, but only read. For objects, const appliesto the referenced object, not to the reference. Const also applies to object methods : it means the method cannot mutate itself, neither its referenced objects. The *self* reference becomes a reference to const object.

Final *Final* qualifier applies to an object reference that cannot reference another object in its lifetime. It doesn't mean the object can't me mutated, unless it's combined with const qualifier.

Arrays Static arrays represent a type constructor : The type *T[]* is a type construced from the type T.

4 Scanning and parsing

4.1 Scanning tokens

- Identifiers : Every lexeme that starts by an underscore or a letter and is followed by underscores, letters or digits is an identifier, unless it's a keyword.
- String Literal : lexeme that is surrounded by quotes "". Escaped quotes inside string should be added in the future.
- Char Literal : same as in C, in single quotes and contains exactly a character (or escaped space, null character etc).
- Number Literal : anything that is a number, be it integer or double. If it is non-integer, it contains a dot followed by digits.
- Bool Literal : true or false.
- Punctuation : Comma, semicolon, dot, arrow (represented by ->)
- Left/Right parentheses ()
- Left/Right brackets {}
- Left/Right square brackets []
- **Keywords :** if, else, elseif, while, for, do, return, function, module, new, struct, class, enum, null.
- **Type qualifiers :** const, final, public, private, protected.
- Operators (boolean, arithmetic, comparison...)