Boboscript

Boris Martin

February 15, 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, and favors *immutable by default* data. 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;
    };
```