bcScript Specification

v 0.1

Contents

0 **Introduction**  
  
 0.1 Design Philosophy

0.2 Structure  
 0.2.1 Analysis  
 0.2.1.1 Lexer  
 0.2.1.2 Parser  
 0.2.2 Intermediate Generation  
 0.2.2.1 Bytecode  
 0.2.3 Execution  
 0.2.3.1 Virtual Machine

0.3

1 **Language Specification**

1.1 Declaration  
 1.1.1 Function  
 1.1.2 Variable  
 1.1.3 Type  
 1.1.3.1 Common Types   
 1.1.4 Compound Type  
 1.1.4.1   
 1.1.5 Namespaces

1.2 Assignment  
 1.2.1 Expressions

1.3 Control Flow  
 1.3.1 If Statements  
 1.3.2 Loop Statements  
 1.3.3 Switch Statements

2 **API Specification**

3

**0 Introduction  
0.1 Design Philosophy**

bcScript is an embeddable scripting language provided as a library of C++ code. Its syntax is intended to be light and intuitive, accessible to non tech users whilst still helpful to an experienced coder.

**0.2 Structure**

The library provides two main functions; the conversion of bcScript code to Bytecode, and the execution of that Bytecode. Provided as C++ code, the library is lightweight and portable. The API provides a means to load plain text script files, as well as use and create Bytecode files that eliminate compilation at runtime. The language comes with native support for the XML format, used in serialisation and data storage. Ease of use and setup are priority, meaning performance and feature set are reduced, and the language is inherently fundamental.

Split into 4 parts, the extent of the bcScript system begins with source code, or at least compiled source code, and ends with bytecode and its compilation. The VM (Virtual Machine) allows the execution of programs, as well as the retrieval of data from the runtime environment. Code within the runtime can be called from scripts and host program alike, as well as the ability to manipulate memory. The VM allows the execution of anonymous code. The API sits atop all these parts and provides methods to load script files, manage VM instances, debug runtimes and provide C++ side access to important functions of bcScript .

**0.2.1 Analysis**

The code is first handled by the Lexer, solely responsible for dividing strings of text into smaller, more discrete strings of text, known as tokens. This small part makes the job of the subsequent Parser much easier, by creating uniformity within the input token stream. Whitespace is stripped, expressions are broken into granular parts, and everything is placed into one of about 60 different groups of token.   
 The resulting stream of tokens is then checked by the Parser for syntactic structure, or the validity of the order of tokens, as well as the semantic check, which is whether a token is valid in its current context.