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 Built in Types  
 1.1.2 Type  
 1.1.2.1 Common Types   
 1.1.3 Compound Type  
 1.1.3.1   
 1.1.4 Functions  
 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 **Standard Library**

4 **Understanding bcScript**

4.1 Introduction

4.2 Compiler Theory

4.3 Coding a Scripting Language  
 4.3.1 Analysis  
 4.3.1.1 bcLexer  
 4.3.1.2 bcParser  
 4.3.2 Intermediate Generation  
 4.3.2.1 bcCodeGen  
 4.3.2.2 bcByteCodeOpt  
 4.3.3 Bytecode Execution  
 4.3.3.1 bcVM

**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. With a small set of fundamental language features,

**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.

**1 Language Specification  
1.1 Declaration**

The Virtual Machine provides a stack for statically typed data

**4 Understanding bcScript  
4.1 Introduction**

Blah

**4.2 Compiler Theory**

**4.3 Coding a Scripting Language**

The language was design also took into consideration the ease in which such a design could be implemented. Complex inheritance hierarchies and higher level constructs are complex to model using only the basic verbs of the processor, so these features were avoided in favour of simple, leveraged tools and a narrower domain of intended use. As mentioned, the language is mainly intended for quick and easy embedding, simple access to script code and data, as well as scripts access to the VM and its runtime data.

**4.3.1 Analysis**

The first step is to take the script and divide it into separate strings, categorized with one of about 60 types. A benefit of taking this step is that our input can be divided into a much smaller set of possibilities. For instance, an Identifier is any name the programmer uses for variables, functions and any other data he may create and use. The range of possible strings we could be presented with is massive, but with a simple set of rules, we can treat all Identifiers uniformly and later check for their validity in the Parser.

**4.3.1.1 bcLexer**

The first step is to take the script and divide it into separate strings, categorized with one of about 60 types. A benefit of this approach is to simply reduce the size of the expressions we will have to code checking for the correct structure of tokens. The API provides us with the class bcLexer with which to perform these operations, as well as a few helper functions to determine which type of token a given string is. All the real legwork is done within NextToken(), so we’ll begin there.

bcToken\* bcLexer::NextToken()

{

//init buffers

bcToken tokbuff;

//get the first char

if(!IncIndex() || done) return NULL;

tokbuff.data=GetChar();

tokbuff.type=DeriveType(tokbuff.data);

tokbuff.col=xindex;

tokbuff.line=yindex;

We begin by creating a bcToken and filling it with the first character of the input stream, which is performed in GetChar().The input is always stored as a two dimensional array of characters, or more precisely, a vector of strings, populated with the data passed to Setup(); GetChar() simply navigates this in a left to right, typewriter fashion.