verbexx Reference

August 24, 2017

Abstract

This document describes the **verbexx** interpreted scripting language, which runs from the Windows 10 command prompt.

verbexx is an unoptimized (exceedingly slow) toy scripting language. It is not industrial strength, nor is it generally useful. **verbexx** resembles a conlang, except it's for computers.

Main objectives of **verbexx** :

- Almost everything is accomplished by verb calls. There are no control statements like **for**, **if**, etc.
- There is no distinction between operators and functions. Verbs can have both left-side and/or right-side positional parameters or keyword parameters (or none).
- Almost all language constructs have to be first-class, since everything is done by verb call. This includes verbs, types, blocks of code (thunks), etc.
- No specialized statement-specific syntax is allowed, since verb calls do everything. This allows the hand-coded DIY parser to be trivial.
- In many cases, sigils are needed to distinguish between verbs, parameters, and keywords.

Table of Contents

List of Examples

Preface

This section contains information that is not directly related to the main **verbexx** description.

When it happened

Coding started in October 2013, and continued on and off until the present. Hopefully, development will continue in the future.

How it happened

verbexx started out as a command file reader, that could invoke complex mapmaking commands, with many positional and/or keyword parameters. From the very beginning, these parameters could be checked for valid types and value ranges. There were no variables, control statements, built-in operators, or user-defined functions.

Variables and Verbs: Soon, it became obvious that variables were needed, along with flow-of-control facilities, built-in operators (like add, subtract, etc.), and then user-defined functions. There was no distinction between operators (like **+** or **=**) and functions, so everything was treated as a verb, no matter how simple or complex the parameters. Pretty soon, first-class verbs were added along with closures, then fixed-length aggregate types.

Control statements: All throughout, the syntax had to be extremely simple, so that only a trivial parser was required. When building the AST, the parser couldn't engage in guesswork or trial and error. Everything about the syntax had to be immediately obvious at first glance. There could be no special syntax for control statements – everything had to be done with verb invocations. Fortunately, keyword parameters for verbs were supported, so control statements (like **@IF**, or **@DO**) could be implemented as verb calls, where some of the keyword parameters (**then:** or **else:**) were a passed-in chunks of code. Variable declarations also had to be done (at run time) using verb calls (**@VAR** or **@CONST**), rather than special syntax.

Sigils: To keep the parser simple, there had to be a way to easily distinguish verb names, variable names, keyword names, and **@GOTO** label names before any variables or user verbs were defined, so sigils were introduced. The **@** leading sigil marked an identifier as

a verb. A `:` leading sigil marked a label, a trailing `:` sigil marked a keyword name, and the leading `$` sigil marked a variable. The default interpretation of an normal-looking identifier is as a variable, so the `$` isn't needed often, but the `@` is needed on all verbs with alphabetic names that could be mistaken for variables. For verbs like `+` or `=`, the default interpretation is as a verb, so no `@` is needed. However, to pass something like `=` as a first-class function argument as a parameter to another function, the leading `$` sigil is required (example: `$=`). Eventually, sigils were added to parenthesis, to indicate how the results of an expression were to be treated when making the AST. The default is to treat a plain expression (like `(a+b+c)`) as an ordinary value. To treat the results as a verb instead, use the `@` sigil before the open parenthesis, and after the close parenthesis. To treat the expression results as a keyword name, use the `:` sigil before the open parenthesis, and after the close parenthesis.

Exceptions: Special verbs like `@GOTO`, `@RETURN`, or `@BREAK` required some sort of "exceptional" return action capability, so an xception facility had to be added. Soon, a need arose for functions and expressions to return multiple values, so this feature was also added. Eventually, the "language" overshadowed the mapmaking aspect of the program, and became the main focus of development activity.

Name changes: Originally, the name of the interpreter executable was simply `ex.exe`, but suddenly one day, the antivirus/anti-malware package started objecting to the name `ex.exe`, and wouldn't allow it to run at all (except for a few hours right after it was erased and recompiled). `ex.exe` was renamed to `verbex.exe` to avoid this problem. Before sending up to Github, the name was changed (again) to `verbexx` , so as to not conflict with other software on the internet. This renaming is similar to what happened when the language Rex was renamed to Rexx before release.

Why it happened

There appears to be a severe shortage of amateur-designed programming languages, so `verbexx` was created to help alleviate this shortage.

Actually, during retirement (after 47+ years working as a programmer), I missed coding, so I started a hobby map-making project in C++. `verbexx` was the end result. Just as language-oriented folks often create their own conlang, many computer folks make their own programming language, just for fun.

1 Introduction

1.1 Script file processing overview

The **verbexx** interpreter executes text files (usually of type .txt) which contain expressions, statements, preprocessor controls, comments, etc. First, the lexer component reads the input files and converts them to a token stream. The preprocessor accepts this token stream, actioning any tokens of direct interest to the preprocessor and passing others through to the main parser. The main parser builds the AST from the token stream, which is then executed by the interpreter component of **verbexx** during the run phase. The lexer, preprocessor, and parser can also be called again to parse strings during the run phase (**@PARSE**) verb).

1.2 Hello World! example

Here is the classic program to print **Hello World!** to the Windows console:

```
1   @SAY "Hello, World!";
```

Example 1: Hello, World!

1.3 Main features

1. There is no distinction between operators and functions. Everything is considered a ”verb”. Verbs can be prefix, postfix, infix, or without arguments. Verbs can be defined to accept zero, one, or many arguments on both the left and right sides.

   Because function invocation syntax is the same as operator invocation syntax, and because functions usually have names that look like variables, the leading **@** is needed to mark an identifier as a verb, rather than its default interpretation as a variable. Because keyword parameters also look like variables, the trailing **:** sigil is needed to mark keywords. Example:

```
1   @VAR ident1 init:1;
```

```
@VAR ident2;
ident2 = 2 * ident1;
@SAY ident1 ident2;
```

Example 2: Sigil usage

Here, the verbs `@VAR` and `@SAY` require the `@` sigil so they will be interpreted as verbs. The `=` and `*` verbs require no sigil, since things that look like operators are interpreted as verbs by default. Also, `ident1` and `ident2` don't need sigils, since they are values, which is the default interpretation of things that look like identifiers. The identifier `init:` needs the trailing semicolon sigil, so it is interpreted as a keyword parameter rather than as a value. Numeric literals like `1` and `2` are always interpreted as values, so sigils are never used with them.

2. Verbs can have positional and keyword arguments on both the left and right sides (right side keyword arguments are far more common).

3. Verbs can be defined with either dynamic or lexical scope (the default). Dynamic scope is used mainly with verbs that get passed chunks of code that access variables in the caller's stack frame.

4. Almost everything is accomplished through verb calls. This includes conditional and looping "statements", variable and verb definitions, and more typical things like variable assignment, arithmetic operations, etc. For example:

```
@IF (x >= 0)
    then:{x}
    else:{-x};
```

Example 3: @IF usage

In this example, `@IF` is a verb that acts like the more usual `if` statement often found in other languages. The `then:` and `else:` keyword arguments are used to supply the code blocks normally found after the `then` or `else` keywords in other languages. Note that a semicolon is required at the after the whole verb invocation, unlike many other langauges. This semicolon is very easy to forget.

5. Most things are first class, meaning they can be assiged to variables, passed as arguments to functions, and returned from functions. This includes:

- variables and literals
- functions and closures
- blocks of code (as seen in the `@IF` example above).

6. Expressions and verbs can have multiple return values.
7. **verbexx** does not support:
    - object-oriented programming, classes, inheritance,mixins, decorators, multiple dispatch, etc.
    - aspect-oriented programming
    - proper functional programming, with pattern matching, immutable data, lazy evaluation, monads, continuations, currying, automatic tail call optimization, term rewriting, functors,etc.
    - regular expressions
    - ranges, generators, and coroutines
    - concurrency, multithreading, green threads, channels, fibers, etc.
    - generic programming and templates
    - type inference
    - namespaces
    - enumerations
    - macros
    - debuggers and IDEs
    - foreign function APIs
    - file I/O (other than writing to the console)
    - extended high-precision/high-range floating point and integer values

2 Running from the command line

2.1 Path and DLL setup

2.2 Command prompt

From the Windows command prompt, enter:

```
1   verbexx filename.filetype arg1 arg2 arg3 ...
```

Example 4: Invocation from command prompt

- `verbexx` is the name of the executable file (`verbexx.exe`).
- `filename.filetype` is the name of the file containing the main source code. It can imbed additional files. `filetype` is usually `.txt`.
- `arg1 arg2 arg3 ...` are optional command line arguments that can be queried during execution.

3 Brief tour of verbexx

3.1 Scalar values

The following types of scalar values are supported:

integer integers signed and unsigned, 8-bit, 16-bit, 32-bit, and 64-bit. The following **verbexx** types are predefined:

- **INT8_T** – 8-bit, signed – implemented using C++ **int8_t**
- **INT16_T** – 16-bit, signed – implemented using C++ **int16_t**
- **INT32_T** – 32-bit, signed – implemented using C++ **int32_t**
- **INT64_T** – 64-bit, signed – implemented using C++ **int64_t**
- **UINT8_T** – 8-bit, unsigned – implemented using C++ **uint8_t**
- **UINT16_T** – 16-bit, unsigned – implemented using C++ **uint16_t**
- **UINT32_T** – 32-bit, unsigned – implemented using C++ **uint32_t**
- **UINT64_T** – 64-bit, unsigned – implemented using C++ **uint64_t**

Arithmetic operations on these numbers work the same as with C or C++, in regards to overflow, wrap-around, etc.

float binary floating point, 32-bit and 64-bit. The following **verbexx** types are predefined:

- **FLOAT32_T** – 32-bit, implemented using C++ **float**
- **FLOAT64_T** – 64-bit, implemented using C++ **double**

Arithmetic operations on these numbers work the same as with C or C++, in regards to floating point precision, floating point range, floating point overflow, floating point underflow, NaNs, floating point rounding mode, etc.

boolean boolean **TRUE**/**FALSE** values – these are implemented using C++ **bool** data type.

empty empty – this a unitialized or non-existent value

3.2 Non-scalar values

The following types of non-scalar data values are supported:

string This is a variable-length string of UTF-16 characters. These are implemented using **std::wstring** containers in C++ STL.

vlist This a normally parameter list for a verb, but can also be used as a general data type. It contains both positional values and keyword values. It is implelented using a C++ STL **std::vector<...>** container for the positional values, plus a **std::multimap<std::wstring,...>** container for the keyword values. Note that duplicate key values are allowed. A verb invoation has two **vlist**s, one for the left side arguments, and another for the right side arguments.

slist This is a chunk of code. It contains zero or more statements. It may contain nested **slist**s. The outermost code block of the script file is itself an **slist**. Any nested **slist** is enclosed in braces (example **{statements}**). **slist**s are first-class values and can be stored in variables, passed to and returned from functions, etc.

The AST produced from the parser is the outermost **slist**, and the **@PARSE** verb can be called during execution to parse a **string** into a nested **slist**, which can be executed or saved away.

Note that **slist**s are not by themselves closures – any variable usage in an **slist** refers to its enclosing scope. Verbs that accept **slist** parameters need to be definew with dynamic scope arther than the default static scope, if the **slist** is to be executed.

Also note, that the outermost (unbracketed) **slist** is automatically executed when the run phase starts. This is one of those rare cases where something in **verbexx** happens without an explicit verb call. However, nested slists need to be explicitly executed by some verb (like **@IF**, **@DO**, etc.).

> Also note, that nested **slist**s do not automatically have their own scope. If an **slist** needs to run in its own scope, that scope must be created by the verb that executes the **slist**. Of course, there is a top-level scope automatically created for the outermost unnested **slist**.

array

struct

function

type

3.3 Verb invocation

3.4 Console output

3.5 Expressions

3.6 Statements and slists

3.7 Built-in verbs

3.8 Verb definitions

3.9 Multiple return values

3.10 Code blocks

3.11 Control ”statements”

3.12 Imported DLLs

4 Lexical processing

4.1 Input stream encoding

4.2 Character set

4.3 Digraphs

4.4 Source code partitions

4.4.1 Comments

4.4.2 Preprocessor text

4.4.3 Everything else

4.5 Output tokens

4.5.1 Sigils

4.5.2 Stand-alone punctuation tokens

4.5.3 Numeric literals

4.5.3.1 String literals

4.5.3.2 Simple strings

4.5.3.3 Strings with escapes

4.5.4 Non-whitespace strings

4.5.5 Identifiers

4.5.6 Operator names

5 Preprocessing

6 Parsing and syntax

7 Execution phase

Index