BespON
Bespoken Object Notation

Geoffrey M. Poore

January 4, 2016

Abstract

BespON is a text-based, human-friendly data serialization format. It is primarily designed for working with lightweight markup languages, with an emphasis on representing literal text with a minimum of escapes. Its type system is a slight superset of JSON's, with support for arbitrary user-specified types. BespON syntax comes in three forms: compact whitespace-independent, indentation-based, and INI-style.

Contents

1 Motivation

BespON grew out of other projects that involve markup languages. I am currently designing a lightweight markup language, and need a data serialization format for creating config files and templates. I also realized that it would be ideal if this same serialization format could be used within the markup language for specifying metadata and settings.[1]

I am also the author of the PythonTEX package for LATEX. The package has been using an ad-hoc format for passing data between LATEX and Python, and would benefit from a more standardized approach.

Why not use an existing serialization format?

JSON

Strings must be quoted, with backslash-escapes for double quotation marks, backslashes, and newlines, amongst other characters. This is not friendly for humans writing or reading extended passages of text, particularly when the text involves a markup language. The markup language will likely have some sort of escape mechanism, which if it involves backslashes will be escaped in the translation to JSON, adding unnecessary visual noise.

YAML

As an indentation-based format, YAML allows for blocks of literal, unescaped text. This is an improvement over JSON.

Unfortunately, YAML's treatment of literal text blocks leaves something to be desired. A text block is specified by a pipe character `|`, followed on the next line by the text, indented to an appropriate level. If all lines in the text block itself are indented, the indentation of the block relative to the current YAML indentation level must be manually specified with an integer in the range 1–9. The indentation must also be manually specified if the first line of the block is indented more than any following line, since YAML does not read ahead to find the line with the least indentation, but rather bases the default indentation of the block on the first line. If a text block ends with empty lines, as it very well might in a template context, keeping these must be manually specified since the default is to discard all trailing empty lines. A typical YAML text block, with an overall indentation of two spaces but a first line indented by four spaces, and with a trailing empty line, might look like this (visible spaces):

```
key1:␣|6+
␣␣␣␣␣␣␣␣␣␣First␣line,␣indented␣four␣spaces
␣␣␣␣␣␣␣␣Second␣line,␣indented␣two␣spaces,␣followed␣by␣empty␣line

key2:␣...
```

Note that the `6` following the pipe indicates that the entire block is indented by a total of 6 spaces relative to the parent node (`key1` indentation level). The number after the pipe does *not* indicate the overall indentation within the actual text block, or the total indentation of the first line of the text; it is based on the position of the text within the YAML file. The `+` indicates that trailing empty lines are to be kept.

Together, the manual specification of indentation level and trailing line treatment mean that if the horizontal indentation, or vertical spacing, of the YAML file is adjusted, then the meaning of the content will change. Thus, while YAML can represent literal text blocks, significant mental overhead is required in writing, reading, and editing.

[1] This is inspired by Pandoc's use of YAML for specifying metadata within a markdown document.

TOML

With its multiline strings, TOML is also an improvement over JSON. In particular, its multiline string literals, delimited with three single quotation marks `'''`, are a significant improvement. Unlike the YAML case, indentation and trailing empty lines are immediately obvious due to the closing quotation marks. No manual specification of indentation or empty line behavior is required.

When working with markup languages, it is quite possible that the three quotation marks `'''` will appear within text. In this case, the user must fall back on escaped strings. This is TOML's primary weakness.

The TOML approach comes closest to what is needed for working with markup languages. While it is ideal in most circumstances, it fails to eliminate the need for escaping completely. What is needed is a literal string delimiter that can adjust to the contents of the text being delimited. It would be possible to use heredoc syntax or C++ raw string literal syntax or Perl quoting, but that would result in significant user-dependent variation. A simpler approach would be to take inspiration from lightweight markup languages, like the fenced code blocks in MultiMarkdown and Pandoc Markdown. That is, have a data serialization format *for* markup inspired *by* markup. Enter BespON.

2 File format

The default encoding for BespON files is UTF-8. The use of a UTF-8 byte order mark (BOM) is strongly discouraged, but it should be checked for and discarded.

BespON libraries are encouraged to support an encoding option for use with legacy applications or in situations in which UTF-16 or UTF-32 are desirable. If a library supports UTF-16 or UTF-32, it should correctly handle BOMs. When an encoding other than UTF-8 is used, it is the responsibility of the sender and receiver to agree on an encoding and act appropriately.

By default, some characters may not appear literally in a BespON file. They may only appear within escaped strings, which use `\xHH` (8-bit), `\uHHHH` (16-bit), and `\UHHHHHHHH` (32-bit) hex notation, as well as the shorthand sequences `\\`, `\'`, `\"`, `\a`, `\b`, `\e`, `\f`, `\n`, `\r`, `\t`, `\v`.[2] All characters with Unicode category "Other, Control" (Cc)[3] must not appear literally by default and must result in an error, with the following exceptions. These characters must always be allowed as literals:

- Line feed U+000A `\n`
- Carriage return U+000D `\r`
- Horizontal tab U+0009 `\t`

For the purposes of line termination, `\n`, `\r`, and `\r\n` should be treated as identical. Implementations may save BespON files with `\r\n` in a system-dependent manner, but using `\n` is preferred.

There are five characters that may optionally be allowed as literals. If these are allowed as literals, they must receive special treatment; if they are not allowed, they must produce errors if they appear. None of these are currently required, because the first three are primarily useful in legacy applications (and are part of the Unicode Cc category), while the last two are relatively recent and not in common use.

[2] Note that some of the shorthand sequences represent characters that are allowed to appear literally. These escaped forms are only used in escaped strings. Which quotation mark escaping is necessary depends on the string delimiters. The escape `\/` should be allowed for cases when similarity to JSON is desired. A library should recognize all of the short escaped forms in reading, but may optionally use `\x`, `\u`, or `\U` forms instead in text it emits.

[3] "Other, Control" includes the range U+0000 to U+001F (ord 0–31), the character U+007F (ord 127, "DELETE"), and U+0080 to U+009F (ord 128–159).

- Form feed U+000C `\f`
- Vertical tab U+000B `\v`
- Next line U+0085 (NEL)
- Line Separator U+2028
- Paragraph Separator U+2029

If an implementation allows these as literals, they should be treated as line terminators for the purposes of parsing, in accordance with the Unicode Newline Guidelines (Unicode Technical Report #13). They should also be preserved within strings, rather than replaced with `\n`, so that round-tripping is possible.

An implementation may have an option that allows some or all of the remaining "Other, Control" characters to be enabled as literals. However, this must not be the default. When using such an option, special care should be taken to avoid security risks. Control characters might be easily overlooked when they are allowed in unescaped form.

The sequence `%!bespon` may be used at the beginning of the first line of a BespON file to indicate the format. In the future, optional arguments that customize parsing or other features may be allowed on the first line, following the `%!bespon` sequence. If the sequence `%!bespon` is detected on the first line of a file, and if it is followed by anything other than whitespace characters, an error must be raised unless (1) the BespON spec has been expanded to include parser directives, (2) the parser supports directives, and (3) the content following the opening sequence can be parsed. The sequence `%!bespon` may appear multiple times within a BespON file that contains multiple datasets, as a delimiter between them. That is, concatenated BespON files are valid. The sequence `%!bespon.eof` may be used in the future in cases in which an end-of-file delimiter is desirable (for example, parsing a log file that may still be open for writing). The sequences `%!besponb`, `%!bespon.bin`, and `%!bespon.binary` are reserved for possible future use in a binary-only variant of BespON.

3 Special characters

BespON uses some characters for special purposes. The set of special characters has purposely been kept as small as possible, so that there is less to consider when escaping. Typically only the percent symbol `%`, parentheses `()`, semicolon `;`, equals sign `=`, and quotation marks `"` and `'` always have a special meaning that may require escaping. In special contexts, the slash `/`, space and tab, plus sign `+`, greater-than sign `>`, square brackets `[]`, period `.`, and underscore `_` also have meaning, but usually in such a way that extra escaping is not required.

BespON allows data to be represented in three forms: a compact form, somewhat analogous to JSON; an indentation-based form, somewhat analogous to YAML; and an INI-style form, somewhat analogous to TOML. Some characters only have special significance in the indentation-based and INI-style forms.

3.1 Comments – percent sign and fullwidth percent sign (and trailing slash and trailing fullwidth solidus)

Single-line comments begin with the percent character `%` (U+0025). A space between the `%` and the beginning of the comment is encouraged.

Multiline comments begin with three or more percent characters, and continue until an identical group of percent characters is found. The closing group of percent characters must be followed immediately by a slash `/` (U+002F). This allows the beginning and end of a long comment to be easily distinguished, even in the absence of syntax highlighting. In multiline comments, the delimiters should be

on lines by themselves. This makes multiline comments more quickly distinguishable from single-line comments.

Example of comments:

```
% Single-line comment

%%%
Multiline comment
that goes on for two lines
%%%/
```

The percent character `%` is chosen for the comment character because it is rare at the beginning of a string. It is used as a comment character in TEX and PostScript, so there is precedent. The hash character `#` could be a logical choice, but given its widespread use as a comment character and in hashbangs and hashtags, using it as a comment character would increase the number of strings that require escaping. The hash character is also common in lightweight markup languages as a header or numbered list indicator. C-style slashes `//` could require that root path names be escaped, and would not allow the clear distinction between opening and closing delimiters for multiline comments.

The fullwidth percent sign ％ (U+FF05) and fullwidth solidus ／ (U+FF0F) may be used instead of the percent sign and slash when working with CJK characters. Otherwise, their use is discouraged.

The sequence `%!bespon` is special. It may be used at the beginning of the first line of a BespON file to indicate the format. Currently, this sequence must not be followed by anything but whitespace. In future versions of the BespON this sequence may optionally be followed by arguments that customize parsing or other features.

The same sequence `%!bespon` may appear multiple times within a file as a delimiter between separate datasets. That is, concatenated BespON files are valid. BespON libraries should provide one mode for parsing BespON files that contain only a single dataset. This mode should raise an error if the sequence `%!bespon` is encountered anywhere other than the first line, with the exception that `%!bespon.eof` must always be valid if it is at the end of the file once trailing whitespace is stripped. Libraries should provide another mode that is used to parse files that contain multiple datasets. This should make all datasets accessible via a list or array, or alternatively iterate over individual datasets. This mode may also return a dict mapping dataset names to datasets, in the event that all datasets include metadata that gives each set a name.

3.2 Indentation – space, tab, and ideographic space

Outside of multiline strings, indentation is only significant in the indentation-based and INI-style forms; in the compact syntax, it is ignored.

Indentation is defined as the space (U+0020), horizontal tab (U+0009), and ideographic space (U+3000). The space is the preferred form of indentation. Four spaces per indentation level is encouraged. Use of the tab is discouraged, but it is allowed for convenience. The ideographic space may be used when working with CJK characters to provide better visual alignment of characters. Otherwise, its use is discouraged.

In determining indentation level, the space, tab, and ideographic space are always treated as distinct characters. Tabs and ideographic spaces are never treated as equivalent to some number of spaces. When a line of text is indented, its indentation characters must exactly match those of the previous line of text, except for those that give the line a greater indentation than the previous line. Any indentation mismatch must result in an error.

In some cases, indentation is based off of a mix of indentation characters and other characters. For example, the first line of a list item might be indented with spaces and a plus sign:

```
␣+␣List␣item
␣␣␣continues␣on
```

If the plus sign is preceded or followed by a space or ideographic space, it is counted as a character (equivalent to a space or ideographic space) for indentation purposes. If ideographic spaces are used for indentation, then the fullwidth plus sign ＋ (U+FF0B) should be used instead of the plus sign (U+002B) to ensure correct visual alignment. If the plus (or fullwidth plus) is both preceded and followed by a tab, then it is ignored in determining indentation level. This approach is applied to all cases in which indentation level must be determined from a mix of indentation characters and a single other character.

There is never an indentation level that mixes indentation characters with more than one non-indentation character. If such a case were allowed, then it would be necessary to normalize Unicode characters and/or use their character categories to determine which code points should be treater as contributing to the indentation level. For a data serialization library, it is simpler avoid this complexity.

3.3 Indented arrays – plus sign and fullwidth plus sign

In non-compact syntax, the plus sign + (U+002B) and fullwidth plus sign ＋ (U+FF0B) are used, together with indentation, to denote the beginning of array items. The fullwidth plus should only be used when ideographic spaces are used for indentation, which should only be done with CJK characters.

A plus (or fullwidth plus) only starts an array when it is the first non-indentation character on a line, is indented relative to the current indentation level, and is followed by one or more indentation characters. An unquoted plus followed by an indentation character may not appear at the beginning of an unquoted string in non-compact syntax, even if it could be determined from the indentation context that the plus is meant as a literal character rather than the beginning of an array item. This promotes clarity and serves as a guard against indentation errors.

When the top level of a dataset is an array, there is no need to indent a plus sign that starts an element, since there is no pre-existing indentation level relative to which to indent.

The first non-indentation character after the plus sign sets the indentation level of the array item content. All content must be indented based on this level.

Logical choices for the beginning-of-array character are the plus +, hyphen -, and asterisk *. The asterisk would be ideal, since it resembles a bullet point. However, it is in common use in markup languages to denote italic/bold (emph/strong), so it may be expected at the beginning of a string. While it would typically not appear at the beginning of a string with a following space (which would require quoting), its appearance could require that writers and readers take a second look to determine whether an array indeed starts. That would add undesirable cognitive load. A leading asterisk could also appear due pointers/dereferencing, etc.

The hyphen is used by YAML and TOML. Like the asterisk, it may be expected as a leading character in many cases. It is used in command-line flags and as an ASCII approximation of an en dash or em dash. A leading dash followed by a space is used in many languages to indicate quoting, so that could also require additional escapes or cause confusion. There is also the issue of visually distinguishing the hyphen from the various sorts of dash characters. Finally, the hyphen does not have the same weight as the plus or asterisk, and thus is more easily overlooked.

This leaves the plus sign for denoting array items. The plus sign is not as commonly used in lightweight markup.[4] In its mathematical and programming use, it rarely appears at the beginning of text. It is more

[4] Org-mode uses it as a strikethrough.

visible than the hyphen. Unlike the asterisk and particularly the hyphen, there are not several other Unicode characters with a similar visual appearance.

Using the plus sign for array items in indentation-based form also provides a nice parallel for dict keys and INI-style form. In these cases, the notation `[+]` is used to refer to the next unused index in a specified array.

3.4 Quoted strings – quotation marks (and trailing slash)

Most BespON strings do not need to be quoted. When strings are quoted, double and single typewriter quotation marks `"` and `'` (U+0022 and U+0027) are used. The two types of quotation marks may be used interchangeably; if a string contains one type, then using the other for quoting is convenient. The fullwidth quotation mark ＂ (U+FF02) and fullwidth apostrophe ＇ (U+FF07) may be used instead in CJK contexts.[5]

There are two kinds of quoted strings. An inline string begins with a group of one or more identical quotation marks, and continues until an identical group of quotation marks is encountered. The closing group of quotation marks may not be bounded on either side by the type of quotation marks it contains. If the first or last non-space (or non-ideographic space) character in a string is a quotation mark of the same type as used in the delimiters, then the outermost space character will be stripped. Thus, `"'"` and `' ' '` would both represent the single quotation mark `'`.

An inline string may be broken across multiple lines. In this case, all lines after the first line with the opening delimiter should have the same indentation. All leading indentation on subsequent lines will be stripped. No line except for the first line, after the opening delimiter, or the last line, before the closing delimiter, may consist solely of indentation characters. The presence of a line of indentation characters in any other location must trigger an error.

Any line breaks within an inline string are converted to spaces, unless the preceding character is a space, in which case they are stripped. An inline string may span multiple lines, but the literal text it represents will always be a single line, without a final newline.

A block string begins with a group of three or more identical quotation marks. Nothing is allowed on the line after this opening delimiter except indentation characters. The actual text follows on subsequent lines; any indentation characters and the line break following the opening delimiter do not contribute to the string. The string ends when an identical group of quotation marks is encountered, unless the opening delimiter is preceded by only indentation characters, in which case the string does not end until a matching closing delimiter with the same indentation is found. If a closing delimiter is not found in this case, then the resulting error message should alert the user to any closing delimiters that matched except for incorrect indentation. The ending quotation marks must be followed immediately by a slash `/` or fullwidth solidus ／. This allows the beginning and end of a block string to be easily distinguished, and provides a guaranteed difference from inline string notation.

Within a block string, all line breaks are preserved. The indentation level is set by the indentation level of the closing delimiter. All lines of text within the block must be indented to at least the level of the closing delimiter; otherwise, an error must be raised. All indentation shared with the closing delimiter is stripped. The opening delimiter is only required to be at the same indentation level as the closing delimiter if the opening delimiter is on a line by itself, but having both delimiters share the same indentation is encouraged. Since line breaks are preserved, the string will end with a newline. If this is not desired, then an additional slash may be added to the closing delimiter, and the final newline will be stripped.

The opening delimiter for both inline and block strings may be identical, though this should be rare in practice. Typically, an inline string should need less than three quotation marks, and the opening

[5] Note that the fullwidth apostrophe will not necessarily resemble a fullwidth version of the single typewriter quotation mark in all fonts; in some fonts, it will appear curled.

quotation marks should almost always be followed on the same line by text. In these cases, the difference between an inline and block string will be immediately apparent. In the rare cases in which an inline string does use three or more quotation marks, it may always be distinguished from a block string by an absence of empty lines and by the absence of one or two slashes / after the closing delimiter. Also, the closing delimiter for an inline string is not required to be on a line by itself.

Inline strings:

```
'A single inline string'
"A string that spans multiple lines
but contains no newlines and doesn't end with a newline"
```

Block strings:

```
'''
A block string with newlines preserved
and ending in a newline.
'''/
"""
A block string not ending in a newline.
"""//
```

Unlike unquoted strings, quoted strings that span multiple lines are not required to be indented relative to the parent element's indentation level, so long as the string does not begin on a line by itself. For example,

```
key = '''
value that
spans multiple lines
'''/
```

This is valid so long as `key` and the opening `'''` are on the same line. If the opening delimiter were on a line by itself, then it would have to be indented to the same level as the closing delimiter. Indenting is encouraged for visual clarity, but it is not required. Parsing is possible due to the delimiters, and in some cases it will be convenient not to have to indent.

In some contexts (for example, streaming), it may be desirable to work with text that contains arbitrarily long runs of quotation marks. In these cases, the opening delimiter should be placed on a line by itself, so that the indentation level of the closing delimiter is set. Then the actual text may be indented by one or more characters relative to this, and this indentation stripped during processing. For example,

```
key =
 |'''
  '''
 |'''/
```

Quoted inline and block strings are only some of the possible string types; others are discussed below.

3.5 Ending delimiters – slash and fullwidth solidus

Slashes / (U+002F) (and fullwidth solidus ／ (U+FF0F) for CJK) only have a special meaning as part of the ending delimiters for multiline comments and block strings. They distinguish the opening delimiters from the closing delimiters in these cases, by appearing as the last character in the closing delimiter. This is reminiscent of the use of the slash in closing HTML tags. In the multiline string case, the slash may be doubled to indicate that the final newline is stripped.

3.6 Collection datatypes and typing – parentheses, fullwidth parentheses, greater-than sign, and fullwidth greater-than sign

In compact form, collection datatypes (array, dict, etc.) are delimited by parentheses `()` (U+0028 and U+0029). When working with CJK characters, fullwidth parentheses `（）` (U+FF08 and U+FF09) may be used instead. Mixing normal and fullwidth parentheses for delimiting a collection is not allowed.

In compact form, whether a collection is an array (indexed by element number) or a dict (indexed by key) is determined by looking for `<key>=<value>` syntax. Serialization formats typically use square brackets `[]` to delimit arrays and curly braces `{}` to delimit dicts. BespON uses parentheses for both cases, with the two being distinguished based on context, to minimize the circumstances in which quoted strings (or other forms of escaping) are required. Square brackets and curly braces are common in markup languages and computing. Having to check all strings for a starting or unpaired bracket or brace would add undesirable cognitive load to working with unquoted strings.

A BespON library may provide an option to use square brackets `[]` to delimit arrays and curly braces `{}` to delimit dicts. However, this must not be the default behavior. If such an option is enabled, then brackets and braces must be treated identically to parentheses in terms of rules for escaping and the conditions under which quoting is required.

A string that contains parentheses must be quoted under the following conditions.

- The string begins or ends with a parenthesis, either opening `(` or closing `)`. An opening parenthesis would otherwise begin a new object, while a closing parenthesis would close the current object.
- The string contains an unpaired parenthesis. Note that the requirement that all parentheses be paired is stronger than the requirement that the string contain an equal number of opening and closing parentheses.

Parentheses are also used to designate types beyond those that may be represented by standard syntax. The syntax `(type)>` is used to designate the type of the following object. Fullwidth parentheses and greater-than sign `＞` (U+FF1E) may be used with CJK characters. Type designations must consist of non-whitespace, non-parentheses characters; no whitespace is allowed between the parentheses, or between the closing parenthesis and the greater-than sign.[6] A greater-than sign is not allowed after a parenthesis that closes a collection type.

BespON specifies some types that must be implemented for `(type)>` syntax. It also specifies several types that may optionally be implemented. A library that does not implement these optional types must raise an error if they are encountered, or if any other unrecognized types are encountered. All official types, both required and optional, are named with a `type.subtype.subsubtype` syntax. In the future, a `type.subtype+subtype` or a `type.subtype=value` syntax may also be used. All official types, both required and optional, will only use the following characters between the parentheses: `a-z`, `A-Z`, `0-9`, and the punctuation characters `+.:=,;_`. The corresponding fullwidth characters must be treated as equivalent.

Libraries may implement types beyond those specified by BespON. These types should be named in `lib:type.subtype.subsubtype` form, where "`lib`" may be replaced with the library name or an abbreviation of it if desired. Libraries may also implement hooks for user-defined types. These should be named in the form `user:type.subtype.subsubtype` or `usr:type.subtype.subsubtype`. `user` and `usr` should be accepted interchangeably. This prevents library and user types from unintentionally colliding with official types.

[6] "Whitespace" should be interpreted as Unicode characters with character property WSpace.

3.7 Typing – greater-than sign

The greater-than sign `>` (U+003E), and fullwidth equivalent `＞` (U+FF1E), is used in indicating typing when it immediately follows a set of parentheses that do not contain any whitespace or parentheses characters.

3.8 Separator – semicolon and fullwidth semicolon

In compact syntax, individual items in a collection datatype are separated by semicolons `;` (U+003B). For example, an array `(1; 2; 3)`. The fullwidth semicolon `；` (U+FF1B) may be used in CJK contexts.

Semicolons and commas are logical choices for item separators. Semicolons were chosen because they appear less frequently in most text and programming contexts, and thus reduce the need for quoted strings. They are also more visible than commas.

In compact syntax, when nothing precedes a separator, it represents the empty string. For example, `(abc;;)` is an array with two elements, the string `abc` and the empty string. The empty string may never be implied in compact syntax, so that the meaning of a separator `;` after the last element in a collection is not ambiguous. Thus, `(abc;)` and `(abc)` are identical arrays containing a single string. The `;)` does not imply a second element containing the empty string.

3.9 Key-value assignment and INI-style form – equals sign and fullwidth equals sign, and slash and fullwidth solidus

The equals sign `=` (U+003D) is used in key-value assignment. For example, `key=value`. The fullwidth equals sign `＝` (U+FF1D) may be used in CJK contexts. The equals sign may be preceded or followed by indentation characters. If the key contains whitespace or special characters that require quoting, then it must be quoted. The equals sign must only be recognized as the separator between a key-value pair when it follows a quoted key, when it is the first equals sign in a run of non-whitespace characters, or when it immediately follows the first run of indentation characters within a run of characters. Any subsequent equals signs are never treated as key-value separators and need not be escaped, with the exception of an equals sign that immediately follows a prior equals sign that is a key-value separator, or an equals sign that would make the string containing it into a key-value pair in the absence of quoting.

In an indentation-sensitive context, the indentation level for `value` must be greater than the indentation level for the first letter of `key`. If `value` is split over multiple lines, the first line upon which it appears after `key` sets the indentation level for subsequent lines, except in the case of quoted strings or similar elements that contain their indentation information in the closing delimiter.

When an equals sign without preceding characters is encountered (for example, `=value`), it will be considered the separator between an empty string key and a value.

As a result of the use of the equals sign in key-value pairs, the text `==` is not allowed. If it meant a key-value pair of the empty string and the equals sign, it would need to be `='='`. If it meant a key-value pair of the empty string with a dict, containing a pairing of the empty string and the empty string, it would need to be `=(=;)` in compact syntax, and would require line breaks in indentation-based and INI-style forms. In a similar way, `==<text>` is not allowed; it would need to be `='=<text>'` or `=(=<text>;)`.

Since the text `==` is not allowed, the text `===` would likewise be invalid. It is repurposed to support an INI-style syntax. A line that begins with `===` must follow it with a path within the data serialization structure. Everything following the line with the `===` will be assigned at this location. For example, everything under the line `===key` would be under `key` in the dict at the current indentation level. A location specified via the `===` syntax is in force until the next use of `===` at the same indentation level. The special sequence `===/` ends all use of the INI-style syntax at the current indentation level. The fullwidth solidus may be used in CJK contexts. An element that is indented less than the opening `===` also ends all use of

the INI-style syntax at the level of the `===`. Uses of `===` at the same indentation level are not nested; using `===/` does not close the current INI-style block while leaving the previous block at the same indentation level open, but rather ends all INI-style syntax at the current indentation level.

3.10 Key paths – period, fullwidth period, square brackets, fullwidth square brackets, plus sign, fullwidth plus sign, underscore, fullwidth low line

In a dict, standard keys set values. BespON also allows for "compound" keys that may be used in a dict to create a series of nested dicts and/or arrays, only setting an actual value at the innermost level. An unquoted key's string is split at periods `.` (U+002E) and fullwidth periods `．` (U+FF0E). Each element in the resulting list is treated as a key to a series of nested dicts. Any key that ends with square brakets `[]` (U+005B and U+005D), or the fullwidth equivalent `［］` (U+FF3B and U+FF3D), containing an integer, is treated as an index in an array. The square brackets need not be separated from previous elements by a period; `key1.key2[0]` and `key1.key2.[0]` are treated as equivalent. The unquoted string `key1.key2[0]=value` would create a key `key1` in the current dict, mapped to a dict in which `key2` is associated with an array, in which the first (zeroth) element is `value`.

If literal square brackets enclosing an integer are desired at the end of a subkey (or as an entire subkey), if periods should not be interpreted as path separators, or if keys (or subkeys) contain whitespace, then keys (or subkeys) may be quoted. Thus, `key1.'key2[0]'` would be two levels deep, with two string keys, while `'key1.key2[0]'` would be a single key. If quoting is required immediately before array indexing, then the indexing must be separated from the quoting by a period. For example, `'key1.key2'.[0]`. This means that in the key part of a dict, it is valid to have two quoted strings immediately adjacent, so long as they are only separated by a period. The period may be thought of as a string concatenation operator, which operates on quoted or unquoted strings. This syntax is not valid in any other context.

When keys refer to arrays, the arrays are always zero-indexed. When an array is assembled piece by piece, the special notation `[+]` may be used to refer to the next unused index in the array (that is, to append an element). The notations `[-1]` may be used to refer to the last element. A trailing pair of square brackets only needs to be escaped when it contains an integer or a plus sign. Negative indices are treated as in Python.

When working with key paths two or more levels deep, the underscore character `_` (U+005F) or fullwidth low line `＿` (U+FF3F), when followed immediately by a period or by square brackets that index an array, refers to the last used key path, with the last element removed. This allows more compact notation. For example,

```
complex_key.property[+] = 1
_[+] = 2
_[+] = 3
```

would be equivalent to the JSON

```
{"complex_key": {"property": [1, 2, 3]} }
```

The underscore does not refer to the full path, because typically the final element will be fully assigned. A literal underscore may be obtained by quoting when this shorthand notation is not desired.

Note that the period, underscore, and square brackets (when enclosing an integer or plus sign) only have special meaning when they occur within a key path. In no other situation, particularly in unquoted strings, do they require any special treatment or escaping.

4 Basic types

BespON defines a number of basic types that must be implemented by a fully comforming library. These types are available without any manual specification of type information.

4.1 String

Since BespON is designed for working with literal text, these are of primary importance. Default strings are pure Unicode string literals, with no escapes of any sort.

Unquoted strings may not begin with a percent sign `%`, parentheses `()`, or quotation marks `'` or `"`. In non-compact syntax, they may not begin with a plus sign `+` followed by indentation characters. In compact syntax, they may not contain semicolons `;` or unpaired parentheses. They may not begin with an equals sign `=`, or have an equals sign separated from the beginning of the string only by non-whitespace characters followed by optional indentation characters. Quoting is also necessary when a string fits a pattern that would cause it to be misidentified as another type.

Unquoted strings may break over multiple lines, but subsequent lines must be indented relative to the parent element. That is, all lines of a wrapped unquoted string should have the same indentation level in an array, but should be indented relative to the key when serving as the value in a key-value pair. Unquoted strings may not contain lines consisting only of whitespace characters. Leading and trailing whitespace characters will be stripped, and any newlines converted to spaces if not preceded by a space and omitted otherwise.

Quoted strings come in inline and block forms as already described.

4.2 Null

Any unquoted string consisting solely of `null` will be converted to null.

4.3 Boolean

Any unquoted string consisting solely of `true` or `false` will be converted to the corresponding boolean.

4.4 Integer

Any unquoted string consisting solely of characters that may be interpreted as an integer. The string representation of an integer cannot involve scientific notation or a decimal point; otherwise, it will be interpreted as a float.

Languages that have an integer type should interpret integers as signed integers or a compatible type. Languages without an integer type should interpret them as floats. At a minimum, support for IEEE 754 binary64 floats is expected. Regardless of the type used, any conversion from string that results in an overflow must result in an error.

Care should be taken when using integer values whose float representation cannot fit within an IEEE 754 binary64 float, since some languages such as JavaScript only have a 64-bit float number type.

A language that does not have an integer type may send BespON integers. However, care should be taken that an integer is actually intended, since in some languages such as JavaScript the default string representation of a float with no decimal part will be a BespON integer.

4.5 Float

Any unquoted string consisting solely of characters that may be interpreted as a float.

At a minimum, support for IEEE 754 binary64 floats is expected. Regardless of the type used, any conversion from string that results in an overflow must result in an error.

Any string consisting solely of `inf`, with or without a sign, or of `nan` will be converted to the corresponding float value.

4.6 Array

An ordered collection of elements. In compact syntax, delimited by parentheses `()` with each element separated by semicolons `;`. Otherwise, indicated by an indented plus sign `+` followed by one or more indentation characters. May also be indicated via key path syntax, with index enclosed in square brackets `[]`.

Regardless of the actual underlying implementation, arrays behave as if they can be accessed by index and allow appending. They behave as Python lists. They are called "arrays" to parallel JSON and TOML, and because of the array-like behavior of Python lists. Arrays may contain elements of multiple types.

4.7 Dict

A mapping of keys to values, typically unordered. Keys may be strings, null, booleans, integers, or floats. However, the use of non-string keys is strongly discouraged, since floats may experience rounding errors on round-tripping, and some languages such as Python do not distinguish between integer and float keys when the two represent the same number.

Libraries may allow keys of other types, but these are the only types of keys a library must support. A library must raise an error if it encounters a key type beyond these that it does not support.

When saving dicts to file, libraries should sort the keys by binary representation in the specified encoding. For example, in typical circumstances, all keys would be encoded as UTF-8, and then the resulting byte strings would be sorted. This provides a fixed key order that makes file diffs meaningful. A library may instead sort the Unicode strings, but since this is locale-dependent and needs to account for Unicode normalization, it is discouraged in the general case.

5 Extended types

BespON defines several extended types that must be implemented by a fully comforming library. These types only available with manual specification of type information.

5.1 Escaped string

Designated by `(str.esc)>`, and the short form `(esc)>`. Delimited like normal quoted strings, with the additional requirement that all characters in the closing delimiter must be unescaped. Ideally, this should be able to be combined with other string types using `(str.subtype+esc)>` syntax.

All characters that are not allowed to appear literally within a BespON file must be represented in escaped form: `\xHH` (8-bit), `\uHHHH` (16-bit), and `\UHHHHHHHH` (32-bit) hex notation. The shorthand sequences `\\`, `\'`, `\"`, `\a`, `\b`, `\e`, `\f`, `\n`, `\r`, `\t`, `\v` are also allowed. For situations in which JSON compatibility is desirable, the escape `\/` should also be allowed. A library should recognize all of the short escaped forms in reading, but may optionally use `\x`, `\u`, or `\U` forms instead in text it emits.

5.2 Stripped newlines string

Designated by `(str.stripnl)>`. In inline strings, instead of conditionally converting newlines into spaces if they would not be preceded by a space, always strip newlines. In block strings, strip all newlines except for those on lines consisting only of indentation characters.

5.3 Unwrapped string

Designated by `(str.unwrap)>`. Treat line breaks in a block string like those in an inline string. That is, conditionally convert newlines into spaces if they would not be preceded by a space, only keeping newlines on lines consisting only of indentation characters. Has no effect on inline strings.

6 Optional types

These types are completely optional. Most will likely not be implemented by a typical library.

6.1 Binary

`(bin.base64)>` and the short form `(bin.b64)>` designates RFC 3548 Base64. The content may be quoted, but this is not necessary given the characters involved.

`(bin.oct)>` designates octal and `(bin.hex)>` designates hexadecimal. In both cases, quoting is allowed but not necessary.

`(bin.raw)>` is reserved for a possible binary BespON variant in the future.

6.2 Ordered dict

Designated by `(odict)>`.

6.3 Set

Designated by `(set)>`.

6.4 Tuple

Designated by `(tuple)>`.

6.5 Array

Designated by `(array)>`. Duplicates the default array.

6.6 Datetime

Designated by `(datetime)>`. Parses a quoted or unquoted string with the RFC 3339 spec.

6.7 IEEE 754

Designated by `(float.binary32)>`, `(float.binary64)>`, `(float.binary128)>`, and by `(float.decimal64)>` and `(float.decimal128)>`. For the binary formats, numbers must always be stored in BespON files in hexadecimal form, to prevent rounding errors. `binary` may be abbreviated to `b` and `decimal` to `d`. Arrays of a given type may be specified via `(array.float.binary32)>` before the array, etc.

6.8 Arbitrary precision arithmetic

Designated by `(int.arb)>`, `(float.binaryarb)>` and `(int.decimalarb)>`. `binary` may be abbreviated to `b` and `decimal` to `d`. Binary floats must be stored in hexadecimal form to prevent rounding errors.

6.9 Copying

With `(copy)>`, a following string is used as a key path, starting at root level. The element pointed to by this path is shallow copied. `(deepcopy)>` may be used for deep copies.

6.10 Special data

The following are used for designating special dicts. They may only be used at the very beginning of a BespON file; this simplifies parsing. The dicts designated with these types are not returned as part of the main dataset. Libraries should provide access to them in another manner, perhaps by wrapping the main dataset in an object with these as some of the properties.

6.10.1 Templating

`(template.var)>` is used to designate a dict that contains key-value pairs that may be substituted into strings. All values must be strings. This must come before any actual substitution; otherwise, an error must be raised. `(str.template)>` is used to indicate that a given string should be treated as a template. Default syntax is `${var}` style, with the braces optional, and doubled dollar sign `$$` for escaping.

6.10.2 Metadata

Metadata may be specified in a dict following `(meta)>`.

6.10.3 Schema

There is not currently a schema system for BespON files. If one is created, schemas may be references or included within a BespON file under `(schema)>`.

7 Data dumping

Because BespON offers three data formats, there is a multitude of ways that even a simple dataset might be represented in BespON format.

Libraries must provide options to serialize data entirely in compact form or in indentation-based form. They may also provide an option to use only INI-style form, although only using this form may be unnecessarily verbose; it may also be less than ideal in other respects.

Libraries are encouraged to provide a way to load data, modify it, and then save it, all while maintaining its formatting and all comments. However, given the added level of complexity that this entails, this is not required. BespON should be thought of first and foremost as a convenient way for humans to get configuration data into computers, rather than a way for computers to rewrite humans' data while keeping the formatting just so. Libraries are also encouraged to provide relatively granular control over the specific form in which data is saved, for working with computer-generated data.

Appendices

Below, JSON, YAML, and TOML examples are provided with corresponding BespON representations.

The JSON example is from Wikipedia. The YAML example is also from Wikipedia. The TOML example is from the TOML GitHub page.

A Compact form

JSON

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

BespON

```
(
  firstName = John;
  lastName = Smith;
  age = 25;
  address = (
    streetAddress = 21 2nd Street;
    city = New York;
    state = NY;
    postalCode = "10021"
  );
  phoneNumber = (
    (
      type = home;
      number = 212 555-1234
    );
    (
      type = fax;
      number = 646 555-4567
    )
  );
  gender = (
    type = male
  )
)
```

B Indentation-based form

YAML

```yaml
---
receipt:     Oz-Ware Purchase Invoice
date:        2012-08-06
customer:
    first_name:   Dorothy
    family_name:  Gale

items:
    - part_no:   A4786
      descrip:   Water Bucket (Filled)
      price:     1.47
      quantity:  4

    - part_no:   E1628
      descrip:   High Heeled "Ruby" Slippers
      size:      8
      price:     133.7
      quantity:  1

bill-to:  &id001
    street: |
            123 Tornado Alley
            Suite 16

    city:   East Centerville
    state:  KS

ship-to:  *id001

specialDelivery:  >
    Follow the Yellow Brick
    Road to the Emerald City.
    Pay no attention to the
    man behind the curtain.
...
```

BespON

```
receipt =     Oz-Ware Purchase Invoice
date =        2012-08-06
customer =
    first_name =   Dorothy
    family_name =  Gale

items =
    + part_no =   A4786
      descrip =   Water Bucket (Filled)
      price =     1.47
      quantity =  4

    + part_no =   E1628
      descrip =   High Heeled "Ruby" Slippers
      size =      8
      price =     133.7
      quantity =  1

bill-to =
    street = '''
             123 Tornado Alley
             Suite 16
             '''/
    city =   East Centerville
    state =  KS

ship-to = (copy)> bill-to

specialDelivery =
    Follow the Yellow Brick
    Road to the Emerald City.
    Pay no attention to the
    man behind the curtain.
```

The `specialDelivery` isn't quite the same as YAML, because YAML will have a final linebreak. The exact equivalent of YAML could be accomplished by using a literal block string combined with `(str.unwrap)>`.

TOML

```toml
# This is a TOML document.

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]

  [servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

  [servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

[clients]
data = [ ["gamma", "delta"], [1, 2] ]

# Line breaks are OK when inside arrays
hosts = [
  "alpha",
  "omega"
]
```

BespON

```
% This is a BespON document.

title = BespON Example

owner =
  name = Geoffrey Poore
  dob = <some number>

database =
  server = 192.168.1.1
  ports = (8001; 8001; 8002)
  connection_max = 5000
  enabled = true

servers =

  alpha =
    ip = 10.0.0.1
    dc = eqdc10

  beta =
    ip = 10.0.0.2
    dc = eqdc10

clients =
  data = ( (gamma; delta); (1; 2) )

% Line breaks are generally OK
hosts = (
  alpha;
  omega
)
```

C INI-style form

YAML

```yaml
---
receipt:     Oz-Ware Purchase Invoice
date:        2012-08-06
customer:
    first_name:   Dorothy
    family_name:  Gale

items:
    - part_no:   A4786
      descrip:   Water Bucket (Filled)
      price:     1.47
      quantity:  4

    - part_no:   E1628
      descrip:   High Heeled "Ruby" Slippers
      size:      8
      price:     133.7
      quantity:  1

bill-to:  &id001
    street: |
            123 Tornado Alley
            Suite 16

    city:   East Centerville
    state:  KS

ship-to:  *id001

specialDelivery:  >
    Follow the Yellow Brick
    Road to the Emerald City.
    Pay no attention to the
    man behind the curtain.
...
```

BespON

```
receipt =    Oz-Ware Purchase Invoice
date =       2012-08-06
=== customer
first_name =  Dorothy
family_name = Gale

=== items
+ part_no =  A4786
  descrip =  Water Bucket (Filled)
  price =    1.47
  quantity = 4

+ part_no =  E1628
  descrip =  High Heeled "Ruby" Slippers
  size =     8
  price =    133.7
  quantity = 1

=== bill-to
street = '''
         123 Tornado Alley
         Suite 16
         '''/
city =  East Centerville
state = KS

ship-to = (copy)> bill-to

specialDelivery =
    Follow the Yellow Brick
    Road to the Emerald City.
    Pay no attention to the
    man behind the curtain.
```

TOML

```toml
# This is a TOML document.

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]

  [servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

  [servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

[clients]
data = [ ["gamma", "delta"], [1, 2] ]

# Line breaks are OK when inside arrays
hosts = [
  "alpha",
  "omega"
]
```

BespON

```
% This is a BespON document.

title = BespON Example

=== owner
name = Geoffrey Poore
dob = <some number>

=== database
server = 192.168.1.1
ports = (8001; 8001; 8002)
connection_max = 5000
enabled = true

=== servers

  === alpha
  ip = 10.0.0.1
  dc = eqdc10

  === beta
  ip = 10.0.0.2
  dc = eqdc10

=== clients
data = ( (gamma; delta); (1; 2) )

% Line breaks are generally OK
hosts = (
  alpha;
  omega
)
```

Another TOML comparison, using a different BespON approach, is below.

TOML

```toml
# This is a TOML document.

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]

  [servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

  [servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

[clients]
data = [ ["gamma", "delta"], [1, 2] ]

# Line breaks are OK when inside arrays
hosts = [
  "alpha",
  "omega"
]
```

BespON

```
% This is a BespON document.

title = BespON Example

=== owner
name = Geoffrey Poore
dob = <some number>

=== database
server = 192.168.1.1
ports = (8001; 8001; 8002)
connection_max = 5000
enabled = true

=== servers.alpha
ip = 10.0.0.1
dc = eqdc10

=== _.beta
ip = 10.0.0.2
dc = eqdc10

=== clients.data
( (gamma; delta); (1; 2) )

% Line breaks are generally OK
hosts = (
  alpha;
  omega
)
```