LSCL - Local Stuff Configuration Language

Version: 0

Licensed by Apache License v2.0

Saint-Petersburg, Russia, 2019.

Contents

List of Figures

Listings

1 Status of this document

This document reflects the first version of LSCL configuration language.

2 Abstract

LSCL is a human-friendly, cross-language, Unicode-based configuration and data serialization language designed around the common native data types. It is designed to be useful as configuration language for local programs on your machine, LSCL contains special features useful for config files, but it's also possible to use it as data serialization language.

3 Introduction

LSCL (Local Stuff Configuration Language) is a human-friendly configuration and data serialization language. This specification contains all information about LSCL syntax and contents, needed for LSCL processing.
LSCL is designed to work with Unicode characters, but all control sequences contain ASCII characters only. LSCL achieves effective data structure by using links, includes, differences and appending containers, so you don't need to write the same things twice.
LSCL is fundamentally built around tree basic primitives as all the most popular data-serialization languages.
There are not so many languages in the world, which stand for storing and transmitting data. LSCL was specially created to work well for common use cases such as: configuration files, log files, interprocess messaging, cross-language data sharing, object persistence and debugging of complex data structures. When data is easy to view and understand, programming becomes a simpler task.
The creation of this language, it's syntax and structure, was inspired by a bunch of data-serialization and programming languages, their pros and cons, history and mistakes. The syntax was motivated by:

- YAML
- JSON
- C++
- System Verilog
- Python

3.1 Goals

1. Easy readable by humans syntax
2. Data is portable between programming languages
3. Machines the native data structures of agile languages
4. Is expressive and extensible
5. Is easy to implement and use
6. Supports references and full path usage to locate node
7. Able to define lists and maps as already existing ones and some difference (added or removed elements)
8. Supports #include of other configuration files in your tree as nodes
9. Supports regex as keys in map structures
10. Designed to use one config as many configs with unsignificant difference

3.2 LSCL relation to JSON

JSON is one of 2 the most popular data serialization languages (JSON and XML), and also easy to read and modify.
XML has the same capabilities as JSON, but it is older and less human-friendly.
LSCL is designed to be backwards-compatible with JSON, so you'll be able to work with your old files using new library.

3.3 LSCL relation to YAML

Author of this standard used YAML for a long time in a big project, but there were not enough features for local YAML usage, as we needed. YAML language is designed more as a language for data-transfers, so it doesn't allow appendig data structures, including files and other features, which you need in your local config.
Other YAML features were mistakes, on the other hand. They were implemented into the language, but were too complex and useless in most cases. For example, YAML allows to use any node as a key in the map, not only scalars.
YAML, of cource, influenced LSCL the most, even this document structure was taken from YAML 1.2 standard.
In LSCL we analyzed advantages and disadvantages of all YAML capabilities and tried to improve them in this standard.
We respect XML, JSON, YAML and other data-representation languages, but we need to move on.

3.4 Terminology

This specification uses key words based on RFC2119 to indicate requirement level. In particular, the following words are used to describe the actions of LSCL processor:

[May] The word may, or the adjective optional, mean that conforming LSCL processors are permitted to, but need not behave as described.

[Should] The word should, or the adjective recommended, mean that there could be reasons for a LSCL processor to deviate from the behavior described, but that such deviation could hurt interoperability and should therefore be advertised with appropriate notice.

[Must] The word must, or the term required or shall, mean that the behavior described is an absolute requirement of the specification.

4 Preview

This section provides quick glimpse into LSCL constructions and features.

4.1 Collections

LSCL's block collections use curved brackets for maps and square brackets for lists. Mappings use colon ":" to mark each key: value pair. Elements of list or map are separated by comma sign "," or newline character. This language uses C-style comments (single and multiline).

```
[
  oc
  toc
  perevertoc
]
```

Listing 1: list

```
{
  babushka: zdorova
  kushaet: kompot
}
```

Listing 2: map

```
[
  { oc1: toc, oc2: toc }
  { perevertoc: zavertoc }
]
```

Listing 3: list of maps

```
{
  oc:  [babushka, zdorova]
  toc: [kushaet, kompot]
}
```

Listing 4: map of lists

```
[
  [babushka, zdorova]
  [kushaet, kompot]
]
```

Listing 5: list of lists

```
{
  oc:  { oc: toc, oc: toc }
  toc: { perevertoc: zavertoc }
}
```

Listing 6: map of maps

It's possible to use links instead of defining repeated objects. Link to the node is first identified by an ampersand sign "&" and the following link name. Link could be referenced with an asterisk "*" thereafter. Link name could be any scalar.
You can use full path to object as well as link to the object. To reference node you can use an asterisk "*" and following full path in brackets: `*("key1"."key2"."key3")`
To reference an element of list you should use index in rectangular brackets instead of key. For example: `*("key1".[6]."key3")`
Here in an example of using different links in LSCL.

```
{
  oc:  &mylink { oc1: toc, oc2: toc }
  toc1: *mylink
  toc2: *"mylink"
  toc3: *(oc)
  toc4: *(oc.oc2)
}
```

Listing 7: using links

4.2 Scalars

LSCL scalars could be written without quotes if they don't contain special chaarcters and newline characters.
Scalars also include two quoted styles. The double-quoted style provides escape sequences. The single-quoted style is useful when escaping is not needed. Quoted scalars can span multiple lines; line breaks are always folded.

```
oc toc perevertoc
```

Listing 8: plain style

```
// {toc} is inside quoted string, so it's
    ↪ just a string

'oc {toc} perevertoc'
```

Listing 9: single-quoted style

```
// Newline character is inserted into
    ↪ string

"oc toc\nperevertoc"
```

Listing 10: double-quoted style

```
// In this case folded line breaks are
    ↪ ignored, spaces at each line end
    ↪ are saved.

'oc toc perevertoc
babushka zdorova
oc toc perevertoc
kushaet kompot'
```

Listing 11: multiline scalar

It's also possible to mark quotes with triangle brackets "<" and ">" .
So, line breaks would be left in string.

```
// In this case all line breaks and spaces are saved

<'oc toc perevertoc
babushka zdorova
oc toc perevertoc
kushaet kompot'>

<"oc toc perevertoc
babushka zdorova
oc toc perevertoc
kushaet kompot">
```

Listing 12: multiline scalar with linebreaks

4.3 Tags

In LSCL some data types could be represented in different ways to improve readability of config.
In addition, some forms of data representation could be detected while processing file and pre-decoded to speed-up config reading process. For example, if you set value 123 without any quotes, it will be treated as integer by default.
If user's code reads scalar value with specific type once, it remembers decoded value and next time returns it to you quicker, without repeated decoding.
It is not necessary for all LSCL-processing libraries to use type-specific value storing, but it is recommended to do so.

```
decimal: 12345
binary: 0b1100
octal: 0o14
hexadecimal: 0x8C1
// + and - signs are also available with all that integers notations
```

Listing 13: integers

```
fixed: 1230.15
canonical: 1.23015e+3
exponential: 12.3015e+02
positive infinity: +.inf
negative infinity: -.inf
not a number: .NaN
```

Listing 14: floating point

```
null_scalar:     ,
null_scalar: NULL
booleans: [ true, false, t, f, T, F, Yes, No, Y, N, +, - ]
string: '012345'
```

Listing 15: miscellaneous

5 Processing

LSCL can be represented both in text form and as tree of LSCL-nodes.
It is designed to be stored in text files, so it supports including nodes from external file into any place of your tree. Processing starts from one "main" file, which will represent root of the tree.

5.1 Commas

Elements in list structure should be separated by commas. The syntax is:

```
[ node_1, node_2, node_3 ]
```

Listing 16: Commas in list (1)

But to prevent errors appear in your config while editing it, it's also made possible to leave commas in the beginning or in the end of list. The following syntax is legal:

```
{
  list 1: [ node_1, node_2, node_3 ]
  list 2: [ node_1, node_2, node_3, ]
  list 3: [ , node_1, node_2, node_3 ]
  list 4: [ , node_1, node_2, node_3, ]
}
```

Listing 17: Commas in list (2)

When elements of list are separated by newline character, it's not necessary to write comma between them. Two next notations are absolutely equal:

```
{
  list 1:
  [
    node_1,
    node_2,
    node_3
  ]
  list 2:
  [
    node_1
    node_2
    node_3
  ]
}
```

Listing 18: Commas in list (3)