Oli Language Documentation

Release 0.0.1

Tomas Aparicio

Sep 27, 2017

Contents

Oli is a minimal declarative language, focused on simplicity, elegancy and with built-in rich features. It is a communitity-driven project based on an open specification

This is the initial Oli language draft reference and specification documentation. It aims to cover all the technical details of the language in order to be useful for implementation and for language features consulting

CHAPTER 1

Project stage

Oli is still a work in progress precarious beta project. Please, consider this before take it as a serious stuff

CHAPTER 2

Document stage

This documentation is a **work in progress** and should be considered unfinished as the language features are still unclosed and under open debating process. Anyone can join to the discussion and help improving Oli

Table of Contents

Overview

About

Oli aims to be a confortable minimal language for general purposes which provides a mix of features from common markup languages and some basic features from programming languages

It has a beutiful, elegant and clean syntax with unobstructive grammar, that allows to use it for multiple purposes, like creating your own DSL. It was designed to be mainly consumed by humans

Rationale

Just for fun :)

Design goals

- Human focused, readability matters
- Keep it simple but provide built-in rich features
- Realiable syntax, avoid ambiguity
- Unobstructive grammar syntax symbol based
- Elegant mix of basic programming languages features

Real use cases

- Create syntax clean and readable configuration files
- Use it as your own DSL for specific purposes
- Store human writable/readable data, like manifest files
- Data interchange format for non-complex schemas structures
- Featured markup-like language for templating
- Replacement for XML, JSON, YAML, INI...

Features

At high level, here are some of the most relevant language features

- First class primitive types: boolean, number and strings
- List and data hashmaps support
- Data linking and references
- Copy or extend data structures
- Templaiting with string interpolation support

Upcoming features

There are some relevant features for the future 0.2 language version specification

- Indent-based blocks (5)
- Block scope references (18)
- Interpolated code (3)
- Built-in functions for data transformation and formatting (#‘21‘_)
- Math operations (20)
- Date as first-class type (2)

For a detailed, take a look to the Github discussion for more details

Specification

Introduction

This is the initial Oli language draft reference and specification documentation. It aims to cover all the technical details of the language in order to be useful for implementation and for language features consulting

About this document

Versioning

This specification follows the semantic versioning convention to as versioning policy.

Just for clarification, related to this document, will be applied as follows

- Minor specification changes and text fixes will be a patch version
- Syntax new feature addon o removement or language logical feature is a minor new version
- General changes with stable document releases will be mayor versions

Syntax examples

Abstract syntax code examples are defined based on context-free grammar EBNF-like metasintaxis

File extension

Just for convention, as it's obvious, the proposed file extension for Oli document is `oli`

MIME Type

There are three different MIME types that can be used to represent the Oli language documents:

- text/oli
- application/oli
- text/oli-template

Character encoding

Oli processor must support the UTF-16 and UTF-8 character encodings

On output it is recommended that a byte order mark should only be emitted for UTF-16 character encodings.

Basics Concepts

Document

Any portion of code that is passed to the Oli processor or compiler, usually as stream of data readed from file in disk, is considered a document

Document is a set of supported syntax valid statements

Linked Data

Oli language must provide built-in support for linking and data consumition (formally variables)

Any type of value in the document can be referenciable, that means any type of primitive value or block (a set of values)

Scopes

Any new document must create it's own referenciable scope context. Oli data references shares the same scope context per document

Just for clarification, if you need to process multiple documents across diferent files, you must merge them before pass it to the Oli compiler

It's under discussion to introduce support for block scopes contexts in the next Oli version

Mutability

Data mutability must not be implemented by norm except in `blocks` and ‘‘lists‘ data types

For example, you could create a reference that points to a number, then it can be overwriten with another number, so it must create a new reference that points to the new value

The unique exception, as commented above, are blocks and lists. Blocks are a set of values that can have another blocks or primitives types. In the case that you use the extend or merge block operators, you must change the block values.

Lists can be also modified via the extend or merge block operators In future language version, the `list` type would have native operators to provide mutation, iteration and manipulation

This feature is under discussion and needs more clarification

Types

Oli implementation must provide built-in support for the following types

Boolean

The boolean type references to the following privimitive literal values.

```
booleanLiteral:
  | true
  | false
  | yes
  | no
  ;
```

The *yes* and *no* are semantic alias to *true* and *false* respectively

Boolean literal values are considered reserved keywords

Number

A number literal is either a decimal or hexadecimal integer of arbitrary size or a decimal double

```
numberLiteral:
     NUMBER
   | HEX_NUMBER
   ;
NUMBER:
     DIGIT+ ('.' DIGIT+)?
```

```
   |  '.' DIGIT+
   ;
HEX_NUMBER:
     '0x' HEX_DIGIT+
   | '0X' HEX_DIGIT+
   ;
HEX_DIGIT:
     'a'..'f'
   | 'A'..'F'
   | DIGIT
   ;
```

If a numeric literal begins with the prefix '0x', it is a hexadecimal integer literal, which denotes the hexadecimal integer represented by the part of the literalfollowing '0x'. Otherwise, if the numeric literal does not include a decimal point denotes an it is a decimal integer literal, which denotes a decimal integer

The numeric literal is a literal double which denotes a 64 bit double precision floating point number as specified by the IEEE 754 standard

There are some limitations in environments like JavaScript engines that can't natively handle 64 bit integers Implementations with this limitation could consider a solution and provide a hackable but consistent way to support it

String

A string is a sequence of valid UTF-8 code units. Oli supports two types of string expressions, quoted and unquoted literal

```
stringLiteral:
  (unquotedString | multilineString | singleLineString)+
  ;
```

Quoted

A quoted string literal is a sequence of characters wrapper with double or singles quotes. A string can be either a single line string or a multiline string and must allow escape sequence characters

```
singleLineString:
    '"' characters* '"'
  | "'" characters* "'"
  ;

multilineString:
   '"' ( characters | NEWLINE )* '"'
 | "'" ( characters | NEWLINE )* "'"
 ;
```

Unquoted

A unquoted string literal is a sequence of valid characters.

An unquoted literal expression can have any type of character except the following sequences

```
":" | NEWLINE | comment | "end" | "[" | "]" | "}" | "{"
```

```
unquotedLiteral:
  (characters (~( ":" | NEWLINE | comment | "end" | "[" | "]" | "}" | "{" )))*
  ;
```

Exception: unquoted strings defined inside a lists or block attributes must add the `,` token as reserved, as it's used in these context as statement terminator token helper

Escape sequence

Strings support escape sequences for special characters. The escapes must are:

- \n for newline, equivalent to x0A
- \r for carriage return, equivalent to x0D
- \f for form feed, equivalent to x0C
- \b for backspace, equivalent to x08
- \t for tab, equivalent to x09
- \v for vertical tab, equivalent to x0B

Nil

A reserved word that reprensent a non-existent or empty value. At programming language level usually is represented by the `null` or ‘‘void‘ primitives types

```
nilLiteral:
  nil
  ;
```

List

A list (formally array) type denotes a list of values, which is an integer indexed collection (in future versions)

List can have elements. Elements can be any type of value, that means a boolean, number, string, block or another list, so it can be a multidimensional list

The `list` is considered a mutable data type, as it can be modified via the block extend or merge operators In a future versions is planned to provide native operators to mutate, iterate and transform lists, just for a better aproach

There are multiple valid expressions to create lists, brackets-based and dash-bash

```
listExpression:
  (listBracketsExpression | listDashExpression)
  ;
listBracketsExpression:
  '[' (element ','?)* ']'
  ;
listDashExpression:
  '-' (element ','?)* NEWLINE
  ;
```

Aditionaly, in order to provide a clean way to create first level document lists, a way to define lists is using the double dash operator (--)

```
listDoubleDashExpression:
  '--' (element ','?)* INPUTEND
  ;
```

Is pending a more detailed specification and examples

Block

A block (formally map or associate array) denotes a key-value map of elements

Blocks the main and most common data type of the language. It's used to build the schema tree and structure the document

A block expression consists of zero or more entries. Each entry has a key and a value. Each key and each value is denoted by an expression. Values can be any type of data, that means a boolean, number, string, list or another block

The `block` is considered a mutable data type

```
blockExpression:
  blockIdentifierExpression assignOperator ((blockElement)*)?
  ;
```

Is pending a more detailed specification and examples

Operators

Unary Operators

Anchor

`&`

Anchor operator is used to create a link references in the document. It is defined as a part of a block identifier expression with a string literal

Reference

`*`

Reference operator is used to consum references in the document. It must be a part of a string literal that defines the reference identifier

Logical Not

`!`

The relational not operator is used in conjunction with the assignment operator to define empty blocks

The use contexts of this operator is under discussion. In the future Oli versions, this operator will be probably overloaded

Pipe

|

The pipe operator is used to define in-line elements in block statements. Currently, the purpose of this operator is only to be a recurrent helper when defining nested block elements without requiring to use the *end* terminator token to express the end of the block

In future versions, this operator will be probably deprecated, due to indentation-based blocks will make unnecesary to use it

Dash

–

The dash operator is used to define list in a shortcut way. It is also used in conjunction with the assignment operator to define raw folded blocks

In the future Oli versions, this operator will be probably overloaded

Assignment Not

!:

The relational not operator is used in conjunction with the assignment operator to define empty blocks

The use of this operator is under discussion.

Binary Operators

Assignment

:

The assignment operator is used as block assignment to define block elements

Equal

=

The equal operator is used as compilation hidden block assignment

In the future Oli versions, this operator will be probably overloaded

Relational

>

The relational operator is used in block identifier expressions to express a short and elegant way to define a block alias that has compilation output effect

In the future Oli versions, this operator will be probably overloaded

Relational Raw

`:>`

The relational raw operator is used as block assignment to define a raw block of literals

Assignment Fold

`:-`

The assignment fold operator is used in block statements to define a folded block of string literals

Assignment Unfold

`:=`

The assignment unfold operator is used in block statements to define a unfolded block of string literals

Extend

`>>`

The extend operator is used in block identifier expressions to define the origin block that should extend from

Merge

`>>>`

The merge operator is used in block identifier expressions to define the origin block that should merge from

Tokens

End

`end`

The end token is used as block statement terminator token. It's a reserved keyword

Comma

`,`

Used as statement terminator helper token inside lists or block attribute expressions

Expressions

Comments

Comments expressions can be defined in any part of the document, including as interpolated expressions inside another expressions. Comments must be ignored from the compiler and optionally by the parser implementation. Comments must have no result in the compilation output

The comment token is #. There are two types allowed comment expressions, in-line or block comments. In-line comments are expressed with a # as stament initializer and the terminator token must be end of line. Block comments starts and end with ##. Both comments contents must allow any type of characte, expect #

```
comment:
  (blockComment | inlineComment)
  ;
inlineComment:
  "#" (character)* NEWLINE
  ;
blockComment:
  "##" (character | NEWLINE (~("##")))* "##"
  ;
```

Identifier

Identifier are expressions which defines a name value that will be processed internally by the compiler for multiple purposes. It is use in blocks to define its idenfitier key, in binary expressions or as reference consumition expression.

```
identifier:
    identifierName
  | '"' character* '"'
  | "'" character* "'"
  ;
identifierName:
  (character | NEWLINE ~(keywords | ":" | NEWLINE | comment | "end" | "[" | "]" | "}"␣
↪| "{" ))*
  ;
```

Pending a better deep explanation about identifier expressions use contexts

String Interpolation

String interpolation allows to use references inside string literal chains. It must be preceded by the * token

```
reference:
    "*" identifierName
  | "*" '"' character* '"'
  | "*" "'" character* "'"
  ;
```

Statements

Value Statement

```
valueStatement:
  identifierExpression assignOperators elements (NEWLINE | endToken)
  ;
```

Variable Statement

```
variableStatement:
  identifierExpression "=" elements (NEWLINE | endToken)
  ;
```

Block Statement

```
blockStatement:
  identifierExpression assignOperator elements (NEWLINE | endToken)
  ;
```

Grammar

Reserved Keywords

The following keywords cannot be used as identifiers. Them must be escaped in order to use it inside identifier or unquoted literals expressions

```
end
nil
true
false
yes
no
:
```

Grammar Ambiguities

This section is still a work in progress

Detected grammar or syntax ambiguities will be detailed here as useful considerations to the developers or end user