Storyscript Documentation

Release 0.25.1

Asyncy

Oct 17, 2019

Contents

Welcome to Storyscript's development documentation. These docs are intended for those who want to contribute to Storyscript itself or those working on an execution engine for the language.

CHAPTER 1

Overview

Storyscript is a domain-specific language that uses **EBNF** for the grammar definition and a **LALR** parser. The abstract tree is then compiled to JSON, which will be used by an execution engine to perform the operations described in the original story.

Unlike other languages, compiling and execution in Storyscript are separated. The services are defined only at run-time at the engine's discretion. In a way, you could say that what Storyscript really does is to convert a story in a machine-friendly format.

For example, the Asyncy engine uses services as docker containers, so when a service is encountered "docker run service-name" is executed by the engine.

CHAPTER 2

EBNF

EBNF stands for "Extended Backus-Naur form", and is a language that can define the syntax for other languages. It looks like this:

```
boolean: TRUE | FALSE
number: INT | FLOAT

TRUE: "true"
FALSE: "false"
INT.2: "0".."9"+
FLOAT.2: INT "." INT? | "." INT
```

If you are curious, you can have at look at Storyscript's full EBNF definition with the grammar command:

```
storyscript grammar
```

CHAPTER 3

LALR

LALR stands for "Look-Ahead LR" parser and is what actually parses a story into its tree.

3.1 Getting Started

3.1.1 Installing from Github

1. Fork the main repository from https://github.com/asyncy/storyscript
2. Storyscript is a Python project. The suggested installation is inside a virtual environment:

```
virtualenv --python=python3.6 storyscript
```

3. Change directory and activate the virtual environment:

```
cd storyscript
source bin/activate
```

4. Clone your fork:

```
git clone git@github.com/asyncy/storyscript.git
```

5. Cd in storyscript, and install it:

```
python setup.py install
```

6. Check the installation:

```
storyscript --version
```

7. You have succesfully installed Storyscript! You might need to install development dependencies as well:

```
pip install tox pytest pytest-mock
```

8. Check you can run the tests:

```
pytest
tox
```

You are now ready to start contributing to Storyscript!

3.2 Command line

A command line interface is provided.

3.2.1 Grammar

The grammar command provides a simple way to check the current EBNF grammar:

```
> storyscript grammar
...
 boolean: TRUE| FALSE
 null: NULL
 number: INT| FLOAT
...
```

3.2.2 Lex

The lex command print a list of all the tokens in a story:

```
> storyscript lex hello.story
 ...tokens list
```

3.2.3 Parse

The parse command returns the internal representation of the abstract syntax tree (AST) after the parsing phase:

```
> storyscript lex hello.story
 ...tokens list
```

3.2.4 Compile

The compile command compiles stories:

```
> storyscript parse hello.story
Script syntax passed!
```

A JSON output of the compilation is available:

```
> storyscript parse -j hello.story

{
  "stories": {
    "one.story": {
      "tree": {
   ...
```

It's possible to specify an EBNF file, instead of using the generated one. This is particularly useful for debugging:

```
> storyscript parse --ebnf-file grammar.ebnf hello.story
```

3.2.5 Help

Outputs the command-line help:

```
> storyscript --help
```

3.2.6 Version

Prints the current version:

```
> storyscript --version
```

3.3 Syntax

Reference for the current syntax

3.3.1 Strings

Strings can be declared with single or double quotes:

```
color = 'blue'
color = "blue"
```

Escaping is done with backslashes:

```
funky = ".\"."
```

Templating

Strings can reference existing variables with templating. In curly brace blocks, variables can be referenced:

```
where = "Amsterdam"
message = "Hello, {where}!"
```

3.3.2 Numbers

Storyscript support integer and floating point numbers:

```
n = 1
pie = 3.14
```

3.3.3 Boolean

Boolean types have two states (*true* and *false*):

```
happy = true
sad = false
```

3.3.4 Lists

Lists are a set of elements with guaranteed order:

```
colours = ["blue", "red"]
```

A list can be defined over more lines:

```
colours = [
    "blue",
    "red",
]
```

Elements are accessed by index:

```
blue = colours[0]
```

3.3.5 Objects

An unordered collection of elements, accessable by key:

```
colours = {'red': '#f00', 'blue': '#00f'}
```

Keys can be variables:

```
colour = 'red'
colours = {colour: '#f00'} # equal to {'red': '#f00'}
```

Objects can be access with dot notation or by key index:

```
colours.red
colours['red']
```

3.3.6 Regular expressions

Regular expressions are defined with slashes:

```
pattern = /^foo/
```

Flags are supported:

```
pattern = /^foo/i
```

3.3.7 Arithmetical expressions

An arithmetical operation between values. Addition (+), subtraction (-), multiplication (*), divison (/), power (^), modulus (%) are supported:

```
a = 1 + 2
a = 1 - 2
a = 1 * 2
a = 1 / 2
a = 1 % 2
```

3.3.8 Logical expressions

A logical operation between values. The logical *and*, *or* and the unary negation (*!*) are supported:

```
c = a and b
c = a or b
c = !a
```

3.3.9 Comparison expressions

A comparison between values. The equal (==), not equal (*!*=), greater (>), greater or equal (>=), less (<) and less or equal (<=) relation are supported:

```
foo == bar
foo != bar
foo > bar
foo >= bar
foo < bar
foo <= bar
```

3.3.10 Control flow

Program flow can be controlled with *if* conditional blocks:

```
if foo
    bar = foo
else if foo > bar
    bar = foo
else
    bar = foo
```

3.3.11 Iterating

Iteration over lists can be done with *foreach*:

```
foreach items as item
    # ..
```

And over objects:

```
foreach object as key, value
    # ...
```

3.3.12 Loops

For more flexible looping, a *while* block can be used:

```
while cond
```

3.3.13 Functions

Functions allow to write repeatable sub procedures:

```
function sum a:int b:int returns int
    x = a + b
    return x
```

The output declaration is optional:

```
function sum a:int b:int
    # ...
```

Calling a function requires parentheses:

```
sum (a:1 b:2)
```

3.3.14 Services

Services can be called with a <*service-name*> <*command-name*> <*arguments*>* expression:

```
result = service command key:value foo:bar
```

Arguments with the value equal to the argument name can be shortened:

```
# instead of: service command argument:argument
service command :argument
```

3.3.15 Streams

When a service provides a stream, the service+when syntax can be used. This could be an http stream, a stream of events or a generator-like result:

```
service command key:value as client
    when client event name:'some_name' as data
        # ...
```

3.3.16 Exceptions

Exceptions can be handled with try:

```
try
    x = 0 / 0
```

Exceptions can be caught:

```
try
    x = 0 / 0
catch as error
    alpine echo message:"caught!"
```

Finally can be used to specify instructions that are always executed, regardless of the try's outcomet:

```
try
    x = 0 / 0
finally
    a = 1
```

3.3.17 Inline expressions

Inline expressions are a shorthand to have on the same line something that would normally be on its own line:

```
service command argument:(service2 command)
```

3.3.18 Mutations

```
1 isOdd
```

Mutations can have arguments:

```
['a', 'b', 'c'] join by:':'
```

3.3.19 Comments

```
# inline
```

```
###
multi
line
###
```

3.3.20 Importing

To import another story and have access to its functions:

```
import 'colours.story' as Colours
```

3.4 Compiler

The compiler takes care of transforming the tree to a dictionary, line by line. Additional metadata is added for ease of execution: the Storyscript version and the list of services used by each story:

```
{
    "stories": {
        "hello.story": {
            "tree": {...}
            "services": ["alpine"],
            "functions": {
                "name": "1"
            },
            "version": "0.5.0"
        },
        "foo.story": {
            "tree": {...},
            "services": ["twitter"],
            "version": "0.5.0"
        }
    },
    "services": [
        "alpine",
        "twitter"
    ],
    "entrypoint": "hello.story"
}
```

3.4.1 The compiled tree

The compiled tree uses a similar structure for every line:

```
{
    "tree": {
        "line number": {
          "method": "operation type",
          "ln": "line number",
          "output": "if an output was defined (as in services or functions)",
          "name": ["if assigning a variable, its name"],
          "service": "the name of the service or null",
          "command": "the command or null",
          "function": "the name of the function or null",
          "args": [
            "additional arguments about the line"
          ],
          "enter": "if defining a block (if, foreach), the first child line",
          "exit": "used in if and elseif to identify the next line when a condition␣
↪is not met",
```

(continues on next page)

(continued from previous page)

```
            "parent": "if inside the block, the line number of the parent",
            "next": "the next line to be executed"
        }
    }
}
```

3.4.2 General properties

Method

The operation described by the line.

Ln

The line number.

Next

Next refers to the next line to execute. It acts as an helper, since the original story might have comments or blank lines that are not in the tree, the next line is not always the current line + 1

Parent

The parent property identifies nested lines. It can be used to identify all the lines inside a block. Care must be taken for further nested blocks.

3.4.3 Objects

Objects are seen in the *args* of a line. They can be variable names, function arguments, string or numeric values:

```
{
    "args": [
        {
            "$OBJECT": "<objectype>",
            "objectype": "value"
        }
    ]
}
```

String

String objects have a *string* property. For example, *"hello"* would evaluate to:

```
{
  "$OBJECT": "string",
  "string": "hello",
}
```

If they are string templates, they will also have a values list, indicating the variables to use when compiling the string. For example, *"hello, {path}* would evaluate to:

```
{
  "$OBJECT": "string",
  "string": "hello, {}",
  "values": [
    {
      "$OBJECT": "path",
      "paths": [
        "name"
      ]
    }
  ]
}
```

List

Declares a list. Items will be a list of other objects. For example, *[1, 2, 3]* would evaluate to:

```
{
  "$OBJECT": "list",
  "items": [1, 2]
}
```

However, note that for other types the object types needs to be passed too. For example, *["hello", "world"]* would evaluate to:

```
{
  "$OBJECT": "list",
  "items": [
    {
      "$OBJECT": "string",
      "string": "hello"
    },
    {
      "$OBJECT": "string",
      "string": "world"
    }
  ]
}
```

Dict

Declares an object:: For example, *["key": "value"]* would evaluate to:

```
{
  "$OBJECT": "dict",
  "items": [
    [
      {
        "$OBJECT": "string",
        "string": "key"
      },
      {
```

(continues on next page)

(continued from previous page)

```
        "$OBJECT": "string",
        "string": "value"
      }
    ]
  ]
}
```

Regexp

Declares a regular expression. For example, */^foo/g* would evaluate to:

```
{
    "$OBJECT": "regexp"
    "regexp": "/^foo/",
    "flags": "g"
}
```

Type

Type objects declare the use of a type:

```
{
  "$OBJECT": "type",
  "type": "int"
}
```

Path

A path is a reference to an existing variable:

```
{
    "args": [
        {
            "$OBJECT": "path",
            "paths": [
                "<varname>"
            ]
        }
    ]
}
```

Is more than one *paths* member given, this implies object access to the referenced variable. For example, *a.b* would evaluate to:

```
{
    "args": [
        {
            "$OBJECT": "path",
            "paths": [
                "a", "b"
            ]
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Expression

Expression have an expression property indicating the type of expression and a *values* array with one (unary) or two (binary) expression values. Values can be‘paths‘ or *values* objects:: For example, *a <type> b* would like similar to:

```
{
  "$OBJECT": "expression",
  "expression": "<type>",
  "values": [
      {
        "$OBJECT": "path",
        "paths": [
          "foo"
        ]
      },
      1
  ]
}
```

Storyscript engines must support the following unary and binary expression types.

3.4.4 Arithmetic operations

- *sum* (*a + b*)
- *subtraction* (*a -b*)
- *exponential* (*a ^^ b*)
- *multiplication* (*a * b*)
- *division* (*a / b*)
- *modulus* (*a % b*)

3.4.5 Logical operations

- *and* (*a && b*)
- *or* (*a || b*)
- *not* (*not a*)

3.4.6 Comparison

- *equals* (*a == b*)
- *greater* (*a > b*)
- *less* (*a < b*)
- *not_equal* (*a != b*)

- *greater_equal* ($a >= b$)
- *less_equal* ($a <= b$)

Argument

Argument objects are used in function definition, function calls and services to declare arguments:

```
{
  "$OBJECT": "argument",
  "name": "id",
  "argument": {
    "$OBJECT": "type",
    "type": "int"
  }
}
```

Mutation

Mutation objects are used for mutations on values, and are found only as arguments in expression methods. They are always preceded by another object, that can be any kind of value or a path:

```
{
  "$OBJECT": "string",
  "string": "hello"
},
{
  "$OBJECT": "mutation",
  "mutation": "uppercase",
  "arguments": []
}
```

Mutations arguments follow the same syntax for service arguments and can be found in the arguments list:

```
{
  "$OBJECT": "mutation",
  "mutation": "slice",
  "arguments": [
    {
      "$OBJECT": "argument",
      "name": "at",
      "argument": 2
    }
  ]
}
```

3.4.7 Methods

Expression

Used for expression lines, like sums, multiplications and so on. For example:

```
1 + 1
```

Compiles to:

```
{
    "method": "expression",
    "ln": "1",
    "output": null,
    "service": null,
    "command": null,
    "function": null,
    "args": [
        {
          "$OBJECT": "expression",
          "expression": "sum",
          "values": [
            1,
            1
          ]
        }
    ],
    "enter": null,
    "exit": null,
    "parent": null
}
```

Setting variables

When declaring a variable, or assigning a value to a property the *name* field will be set. For example, a story like:

```
x = "hello"
```

Will result in:

```
{
    "1": {
      "method": "expression",
      "ln": "1",
      "name": ["a"],
      "args": [
        1
      ],
      "next": "<next line>"
    }
}
```

If

Args can be a path, an expression object or a pure value. When part of block of conditionals, the exit property will refer to the next *else if* or *else*.

For example, *if color* would evaluate to:

```
{
  "method": "if",
  "ln": "1",
  "output": null,
```

(continues on next page)

(continued from previous page)

```
  "service": null,
  "command": null,
  "function": null,
  "args": [
    {
      "$OBJECT": "path",
      "paths": [
        "color"
      ]
    }
  ],
  "enter": "2",
  "exit": null,
  "parent": null,
  "next": "2"
}
```

Elif

Similar to *if*. For example, *elif a == 1* would evaluate to:

```
{
  "method": "elif",
  "ln": "3",
  "output": null,
  "service": null,
  "command": null,
  "function": null,
  "args": [
    {
      "$OBJECT": "expression",
      "expression": "equals",
      "values": [
        {
          "$OBJECT": "path",
          "paths": [
            "a"
          ]
        },
        1
      ]
    }
  ],
  "enter": "4",
  "exit": null,
  "parent": null,
  "next": "4"
}
```

Else

Similar to if and elif, but exit is always null and no args are available:

```
{
  "method": "else",
  "ln": "5",
  "output": null,
  "service": null,
  "command": null,
  "function": null,
  "args": [],
  "enter": "6",
  "exit": null,
  "parent": null,
  "next": "6"
}
```


Try

Declares the following child block as a try block. Errors during runtime inside that block should not terminate the engine:


```
{
  "method": "try",
  "ln": "1",
  "next": "2",
  "name": null,
  "function": null,
  "output": null,
  "args": null,
  "command": null,
  "service": null,
  "parent": null,
  "enter": "2",
  "exit": null
}
```


Catch

Declares the following child block as a catch block that would be executed in case the previous try block failed:


```
{
  "method": "catch",
  "ln": "3",
  "output": [
    "error"
  ],
  "name": null,
  "function": null,
  "args": null,
  "command": null,
  "service": null,
  "parent": null,
  "enter": "4",
  "next": "4",
  "exit": "line"
}
```

Finally

Declares the following child block as finally block that is always executed regardless of the previous try outcome:

```
{
  "method": "finally",
  "ln": "5",
  "name": null,
  "function": null,
  "output": null,
  "args": null,
  "command": null,
  "service": null,
  "parent": null,
  "enter": "6",
  "next": "6",
  "exit": null
}
```

Foreach ### Declares a for iteration. For example *foreach items as item* would evaluate to:

```
{
  "method": "for",
  "ln": "1",
  "output": [
    "item"
  ],
  "service": null,
  "command": null,
  "function": null,
  "args": [
    {
      "$OBJECT": "path",
      "paths": [
        "items"
      ]
    }
  ],
  "enter": "2",
  "exit": null,
  "parent": null,
  "next": "2"
}
```

Execute

Used for services. Service arguments will be in *args*. For example, *alpine echo message: "text"* would evaluate to:

```
{
  "method": "execute",
  "ln": "1",
  "output": [],
  "name": [],
  "service": "alpine",
  "command": "echo",
  "function": null,
```

(continues on next page)

(continued from previous page)

```
  "args": [
    {
      "$OBJECT": "argument",
      "name": "message",
      "argument": {
        "$OBJECT": "string",
        "string": "text"
      }
    }
  ],
  "enter": null,
  "exit": null,
  "parent": null
}
```

Function

Declares a function. Output maybe null. For example, *function sum a:int b: int returns int* would evaluate to:

```
{
  "method": "function",
  "ln": "1",
  "output": [
    "int"
  ],
  "service": null,
  "command": null,
  "function": "sum",
  "args": [
    {
      "$OBJECT": "argument",
      "name": "a",
      "argument": {
        "$OBJECT": "type",
        "type": "int"
      }
    },
    {
      "$OBJECT": "argument",
      "name": "b",
      "argument": {
        "$OBJECT": "type",
        "type": "int"
      }
    }
  ],
  "enter": "2",
  "exit": null,
  "parent": null,
  "next": "2"
}
```

Return

Declares a return statement. Can be used only inside a function, thus will always have a parent. For example, *return x* would evaluate to:

```
{
  "method": "return",
  "ln": "2",
  "output": null,
  "service": null,
  "command": null,
  "function": null,
  "args": [
    {
      "$OBJECT": "path",
      "paths": [
        "x"
      ]
    }
  ],
  "enter": null,
  "exit": null,
  "parent": "1"
}
```

Call

Declares a function call, but otherwise identical to the execute method. For example, *sum(a: 1, b:2)* would evaluate to:

```
{
  "method": "call",
  "ln": "4",
  "output": [],
  "service": "sum",
  "command": null,
  "function": null,
  "args": [
    {
      "$OBJECT": "argument",
      "name": "a",
      "argument": 1
    },
    {
      "$OBJECT": "argument",
      "name": "b",
      "argument": 2
    }
  ],
  "enter": null,
  "exit": null,
  "parent": null
}
```

3.5 Error codes

3.5.1 E0001 Unindentified error

An error that can't be identified to a specific error code.

3.5.2 E0002 Service name

A service name contains a dot

```
alp.ine echo
```

3.5.3 E0003 Arguments noservice

No service was found for an argument, usually because there is an indented argument, but the preceding service has not been found

```
alpine
x = 0
    message:"hello"
```

3.5.4 E0004 Return outside

A return statement is outside a function

```
return "hello"
```

3.5.5 E0005 Variables backslash

A variable name contains a backslash

```
my/variable = 0
```

3.5.6 E0006 Variables dash

A variable name contains a dash

```
my-variable = 0
```

3.5.7 E0007 Assignment incomplete

An assignmnent that is missing a value

```
x =
```

3.5.8 E0008 Function misspell

The *function* keyword was misspelt.

```
func hello
    x = 0
```

3.5.9 E0009 Import misspell

The *import* keyword was misspelt.

```
imprt 'cake' as Cake
```

3.5.10 E0010 Import as misspell

The *as* keyword in an import statement was misspelt.

```
import 'cake' a Cake
```

3.5.11 E0011 Import unquoted file

The filename of an import statement is unquoted.

```
import cake as Cake
```

3.5.12 E0012 String opening quote

A string is missing the opening quote.

```
hello'
```

3.5.13 E0013 String closing quote

A string is missing the closing quote.

```
'hello
```

3.5.14 E0014 List trailing comma

A list has a trailing comma.

```
[1,]
```

3.5.15 E0015 List opening bracket

A list is missing the opening bracket.

```
1]
```

3.5.16 E0016 List closing bracket

A list is missing the closing bracket.

```
[1
```

3.5.17 E0017 Object opening bracket

An object is missing the opening bracket.

```
x: 0}
```

3.5.18 E0018 Object closing bracket

An object is missing the closing bracket.

```
{x: 0
```

3.5.19 E0019 Service argument colon

A service argument is missing the colon.

```
alpine echo message
```