ZUMA

Language Specification

Contents

1 Language constructs

1.1 Datatypes and constants

ZUMA is strongly typed.

Following datatypes can be declared using literals:

Bool has one of values `true` or `false`.

Number is a single precision floating point, i.e. f32: `1.5464`.

Point is declared using two numbers inside square brackets like `[4.45,6.06]`.

Color can be declared using sharp followed by hexadecimal value: `#ff00a1`. Additionally few basic colors can be declared by their name: `black`, `white`, `red`, `green`, `blue` or `yellow`.

Text is delimited by double quotes.

We can declare constant *name* with value *literal* using keyword `let`:

```
let <name> = <literal>;
```

1.2 Arithmetic and logical operations

ZUMA is not intended to be general purpose programming language, therefore design goal is to **minimize** amount of supported data types, operations and language constructs.

However for generating complex structures, such as ones consisting of repeated elements (*for loops*) or of parts rendered conditionally (*if statements*), we have to support these language constructs. And these constructs require support for aritmetic (loops) and logical (ifs) operations.

ZUMA understands these aritmetic operations:

Arithmetic addition expressed by symbol `+`.

Subtraction expressed by symbol `-`.

Arithmetic multiplication expressed by symbol `*`.

Division expressed by symbol `/`.

And following logical ones:

Less than expressed by symbol `<`.

Greater than expressed by symbol `>`.

Equality expressed by symbol `==` (double equal sign).

Logical addition (or) expressed by symbol `|`.

Logical multiplication (and) expressed by symbol `&`.

Negation expressed by symbol `!`.

Additionally, left and right parenthesis `( )` can be used to group operations into higher level units and set precedence of operations.

QUESTION: Is it possible to have NO operators precedence and rely on parentheses exclusively?

1.3 Expressions

Expressions are delimited using semicolon.

```
line start = [0,10] end = [25,50] color = #ff00a1;
```

Expressions are following constructs:

- constant declaration
- function call
- scope

1.4 Functions

Function is called using its name followed by list argument assignments, like this:

```
line start=[0,0] end=[15,20] color=white;
```

Here `line` is name of function being invoked, `start`, `end` and `color` are arguments being passed to function. Like every expression it has to end with semicolon.

There is number of predefined functions, mainly concerned with generating graphical primitives.

Support for user defined functions is planned, but not finalized.

QUESTION: We will want to have two kind of functions, one to calculate and *return* some complex value, possibly to be used inside arithmetic or logical operation. Second kind to directly render some graphical object. We will need to distuingish these. Possibly functions to directly render graphics should instead return these (with ability to assign them to constant) and only when invoked without assignment should they be rendered.

1.5 Scopes

Scope is delimited by `{` and `}`. There is list of expressions between braces. Scope is an expression.

1.6 Comments

Single line:

```
// this is comment
```

Part-line / multiline:

```
/* multiline comment */
```

Comments can be nested:

```
/* /* */ */
```

```
/* */ */
```

```
/* /* */
```

Anything inside comments shouldn't break compilation.

2 Predefined functions

2.1 Line

2.2 Rectangle

2.3 Text

3 Notes

3.1 Coordinate system

Origin point is left upper corner. `x` is vertical axis, `y` is horizontal axis.

Therefore `[0,500]` describes upper right corner, while `[500,0]` describes lower left corner.

3.2 Reserved words

Color literals: `black white red green blue yellow`

Boolean literals: `true false`

Pre-defined functions: `line rectangle text`

Constant declaration keyword: `let`

3.3 Architecture

Parser

Abstract Syntax Tree

Evaluation: remove comments, eval variables, ifs and for loops...

Translation: ZUMA IR is basically in-memory OOP model of SVG to be generated. It is result of evaluation step and used to generate SVG.

Generate SVG

3.4 Performance goals

1ms - good

10ms - acceptable

100ms - bad

1000ms - unacceptable