
ArcJS Documentation

Release 0.1

smihica

March 06, 2017

1	Contents	1
1.1	Tutorial	1
1.2	NameSpace reference	17
2	What is ArcJS?	19
3	What is Arc-language?	21

Tutorial

Installation and Usage

from npm

```
$ sudo npm install -g arc-js
```

from source

Install nodejs first.

In mac:

```
# from port
$ sudo port install nodejs
$ sudo port install npm
```

```
# from brew
$ brew install node.js
```

In linux (debian):

```
$ sudo apt-get install nodejs
$ sudo apt-get install npm
$ sudo apt-get install node-legacy
```

Clone this repository and make.

```
$ git clone https://github.com/smihica/arc-js.git
$ cd arc-js
$ npm install
$ make; make test
```

Then go to your project.

```
$ npm install /ArcJS/Repository/PATH
```

Using in node

```
$ node
> var ArcJS = require('arc-js');
> ArcJS.version
'X.X.X'
> var arc = ArcJS.context();
> arc.evaluate('(prn "hello world")');
hello world
'hello world'
>
```

Using on a webpage (JavaScript)

```
$ cp /ArcJS/Repository/PATH/arc.min.js .
$ echo index.html
<html>
  <head>
    <script type="text/javascript" src="arc.min.js"></script>
    <script>
      var arc = ArcJS.context();
      arc.evaluate('(prn "hello world")');
    </script>
  </head>
  <body></body>
</html>
```

When open the webpage then “hello world” will be in console.

Using on a webpage (Arc)

```
$ echo index.html
<html>
  <head>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="jquery.min.js"></script>
    <script type="text/javascript" src="arc-loader.js"></script>
    <script type="text/arc">
      (prn "hello world")
    </script>
  </head>
</html>
```

Playing in REPL

Starting REPL

If you’ve installed via `npm install -g`

```
$ arcjs
arc>
```

otherwise `npm install`

```
$ node_modules/arc-js/bin/arcjs
arc>
```

Atoms

everything following `;` is a comment (until end-of-line)

Symbols

```
arc> t
t
arc> nil
nil
arc> 'a
a
arc> 'u-nk_~o#abc$$$%moemoe
u-nk_~o#abc$$$%moemoe
arc> '|a b c| ;; A symbol has delimiter strings
|a b c|
```

Numbers

```
arc> 0
0
arc> 3.14
3.14
arc> -inf.0
-inf.0
arc> #x10 ;; hexadecimal notation
16
```

Characters

```
arc> #\a
#\a
arc> #\ ;; unicode
#\
```

Escape characters are `#\nul` `#\null` `#\backspace` `#\tab` `#\linefeed` `#\newline` `#\vtab` `#\page` `#\return` `#\space` `#\rubout`

```
arc> #\newline
#\newline
```

Strings

```
arc> "abc"
"abc"
arc> "" ;; unicode
""
```

```
arc> "a\nb"
"a\nb"
arc> "\u000A" ;; unicode
"\n"
```

Cons

```
arc> '(a b)
(a b)
arc> '(a . (b . c))
(a b . c)
```

Regular Expression (using JavaScript's RegExp() internally)

```
arc> #/a/
#<regex /a/>
arc> #/^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$/
#<regex /^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$/>
```

Hash table

```
arc> (table)
{}
arc> (table 'key 'val)
{key val}
```

n=d indicates the number of properties. And to access the key

```
arc> (= tbl (table 'key 'val))
{key val}
arc> (tbl 'key)
val
arc> (tbl 'notfound)
nil
```

If the key is not found nil is returned.

Special syntax for table. { key value } to be (table key value)

```
arc> { :key1 :value1 "key2" "value2" }
{:key1 :value1 "key2" "value2"}
```

keys can be any type.

```
arc> (= tbl {
  :abc "def"
  "GHI" 'jkl
  'mno "pqr"
  10 100
})
{:abc "def" "GHI" jkl mno "pqr" 10 100}
arc> (tbl :abc)
"def"
arc> (tbl "GHI")
```



```

jkl
arc> (tbl 'mno)
"pqr"
arc> (tbl 10)
100
arc> (tbl 'notfound)
nil

```

Tagged

```

arc> (annotate 'my-type (table))
#<tagged my-type {}>

```

Expressions

Arc (and most of all lisp languages) uses S-Expression.

```

arc> (+ 1 2)
3
arc> (+ (/ 1 2) 3)
3.5

```

Binding local variables

To bind local variables, there are `let`, `with` and `withs` syntax.

- `let` binds just 1 value

```

(let var val body)

arc> (let a 10
      (+ a (* a 2)))
30

```

- `with` binds multiple values

```

(with (var1 val1 var2 val2) body)

arc> (with (x 3 y 4)
      (sqrt (+ (expt x 2) (expt y 2))))
5

```

There is also `withs` syntax that can bind sequentially.

```

(withs (var1 val1 var2 using-var1) body)

arc> (withs (x 3 y (* x 10))
          (+ x y))
33

```

You also can use pattern matching in `let` / `with` / `withs`.

```

arc> (let (a b c . d) '(1 2 3 . 4)
      (* a b c d))
24

```

Conditions

There is some condition statements. `if` `when` `aif` `awhen` `case`

`(if condition then else)`

In arc, all non-nil values are truthy.

```
arc> (if 0 'a 'b)
a
arc> (if nil 'a 'b)
b
arc> (if nil 'a)
nil
```

Use `(no x)` to invert the logic.

```
arc> (if (no nil) 'a 'b)
'a
arc> (if (no (odd 2)) 'a)
'a
```

In arc

```
(if a b c d e)
```

is same as

```
(if a
  b
  (if c
    d
    e))
```

is / iso

```
arc> (is 'a 'a)
t
arc> (is '(a b) '(a b))
nil
arc> (iso '(a b) '(a b))
t
```

Binding

using `=`, you can bind value into a variable.

```
arc> (= s '(f o o))
(f o o)
arc> s
(f o o)
```

You also can bind into a place.

```
arc> (= (s 0) 'm)
m
arc> s
(m o o)
```

You can define your own set function by using `defset`.

```
arc> (defset caddr (x)
      (w/uniq g
       (list (list g x)
              `(caddr ,g)
              `(fn (val) (scar (caddr ,g) val))))))
```

Then

```
arc> (= (caddr s) 'v)
v
arc> s
(m o v)
```

to get more information for `defset` read [here](#).

multiple expressions

You can run some expressions sequentially using `do` or `do1`. `do` returns result of the last expression.

```
arc> (let x 2
      (if (even x)
          (do (prn x " is even value !!")
              (* x 10))
          (do (prn x " is odd value!!")
              (/ x 10))))
2 is even value !!
20
```

`do1` returns result of the first expression.

```
arc> (do1 (prn (+ 2 " is even value !!"))
       (prn (+ 3 " is odd value !!")))
2 is even value !!
3 is odd value !!
"2 is even value !!"
```

def

`def` defines new global function into current namespace.

```
arc> (def fizz-buzz (l)
      (for n 1 l
           (prn (case (gcd n 15)
                    1 n
                    3 'Fizz
                    5 'Buzz
                    'FizzBuzz))))
#<fn:fizz-buzz>
arc> (fizz-buzz 100)
1
2
Fizz
...
```

iterate syntaxes

There are a lot of iterate syntax in arc. for each while repeat map

```
arc> (for i 1 10 (pr i " "))
1 2 3 4 5 6 7 8 9 10 nil
arc> (each x '(a b c d e)
      (pr x " "))
a b c d e nil
arc> (let x 10
      (while (> x 5)
        (= x (- x 1))
        (pr x)))
98765nil
arc> (repeat 5 (pr "la "))
la la la la la nil
arc> (map (fn (x) (+ x 10)) '(1 2 3))
(11 12 13)
```

short cut function syntax

[+ _ 10] will be compiled to (fn (_) (+ _ 10))

```
arc> (map [+ _ 10] '(1 2 3))
(11 12 13)
```

mac

```
arc> (mac when2 (tes . then) `(if ,tes (do ,@then)))
#<tagged mac #<fn:when2>>
arc> (when2 t 1 2 3)
3
```

Arc's mac creates legacy macros, so you can create macros that binds variables implicitly.

```
arc> (mac aif2 (tes then else)
      `(let it ,tes
          (if it ,then ,else)))
#<tagged mac #<fn:aif2>>
arc> (aif2 (car '(a b c)) it 'x)
a
```

By using w/uniq, you can create one-time symbols.

```
arc> (mac prn-x-times (form times)
      (w/uniq v
        `(let ,v ,form
            (do ,@(map (fn (_) `(prn ,v)) (range 1 times))
                nil))))
#<tagged mac #<fn:prn-x-times>>
arc> (let i 5 (prn-x-times (++ i) 3))
6
6
6
nil
```

(w/uniq (v1 v2 v3 ...) body) is also OK.

continuation

You can create continuations by using `ccc`

```
arc> (ccc
      (fn (c)
        (do (c 10)
            (err))))
10
;; like yield
arc> (ccc
      (fn (return)
        (let x 0
          (while t
            (let adder
              (or (ccc (fn (c)
                        (= next c)
                        (return x)))
                  1)
                (++ x adder))))))
0
arc> (next nil)
1
arc> (next nil)
2
arc> (next 10)
12
arc> (next nil)
13
```

symbol-syntax

As an arc's function, there are macros that'll be expanded when a symbol matches some patterns. This function named `symbol-syntax`. For example `(car:cdr x)` will be expanded `(car (cdr x))` (If there is `:` in the symbol then expands).

```
arc> (car:cdr '(1 2 3))
2
```

And `~x` will be expanded `(complement x)`

```
arc> (if (~no 'a) 'b 'c)
c
```

You can check the expanded expression of `symbol-syntax` by using `ssexpand`.

```
arc> (ssexpand 'abc:def)
(compose abc def)
arc> (ssexpand '~no)
(complement no)
```

As ArcJS's expansion, there is a function that makes users be able to define arbitrary `symbol-syntax`; `defss`.

For example, lets define new special-syntax that is able to expand `(caadaar x)` or `(cadadadadadar x)` to expressions composed `car` and `cdr`.

```
arc> (defss cxr-ss #/^c([ad]{3,})r$/ (xs)
      (let ac [case _ #\a 'car #\d 'cdr]
```

```
`(fn (x)
  , ((afn (xs) (if xs ` (, (ac (car xs)) , (self (cdr xs))) 'x)
    (coerce (string xs) 'cons))))
#<tagged special-syntax (#<regex /^c([ad]{3,})r$/> 12 #<fn:cxr-ss>)>
```

Then

```
arc> (ssexpand 'caaaar)
(fn (x) (car (car (car (car x)))))
arc> (ssexpand 'cadadar)
(fn (x) (car (cdr (car (cdr (car x)))))
```

So you are able to do this

```
arc> (caddddddddr '(1 2 3 4 5 6 7 8 9 0))
9
```

namespaces

ArcJS has a namespace extension. It works like Clojure's namespace. To create a namespace use `(defns xx)`.

```
arc> (defns A)
#<namespace A>
```

And then you can go into the namespace by using `(ns namespace)`.

```
arc> (ns 'A)
#<namespace A>
arc:A>
```

Or you also can use string or namespace object as its' first argument.

```
;; using string
arc> (ns "A")
#<namespace A>
arc:A>

;; This way is the most commonly pattern.
arc> (ns (defns B))
#<namespace B>
arc:B>
```

As you see, the prompt has been changed to `arc:A>` to describe where namespace we are in now.

To get the namespace that we are in now use `(***curr-ns***)`.

```
arc:A> (**curr-ns**)
#<namespace A>
```

By the way, When you define a variable named like `***VAR***`, You can access same value bound in it wherever you are. In a word, a variable named like `***VAR***` will behave a namespace global variable.

To export names use `:export` like

```
arc> (defns A :export fn1 macro1 fn2)
```

Then, You can access `fn1 / macro1 / fn2` in any namespaces that import namespace `A`. If you don't specify `:export`, every variables in the namespace will be exported.

And To import other namespace use `:import` like

```
arc> (defns C :import A B)
```

Then, You can access values exported in namespace B and C when you go into namespace A. But you can't access values imported in B and C. By this time, namespace B and C must be loaded beforehand.

And there is also `:extend` option.

```
arc> (defns D :extend A)
```

When you use it, you can extend the specified namespace. In the new namespace, you can access all the variables in specified namespace.

How to use on a webpage

Now, Let's see how to work ArcJS on a webpage. First, We will begin with Hello Word. Please create html like following.

```
<!doctype html>
<html>
  <head>
    <script type="text/javascript" src="//code.jquery.com/jquery-1.10.2.js"></script>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="arc_loader.js"></script>
  </head>
  <body>
    <textarea id="holder" style="width:500px;height:600px;"></textarea>
  </body>
</html>
```

```
arc.min.js arc_loader.js
```

Then please add some hello world in ArcJS after `arc_loader.js`

```
...
<script type="text/javascript" src="arc_loader.js"></script>
<script type="text/arc">
(js/log "Hello world !!")
</script>
...
```

As you see, In case of you've loaded `arc_loader.js`, The content in `<script type="text/arc">...</script>` will be run in ArcJS's context on page onload timing. And Of course you can do like this `src="hello_world.arc"` to export arc code as another file. Like this.

```
...
<script type="text/javascript" src="arc_loader.js"></script>
<script type="text/arc" src="hello_world.arc"></script>
...
```

`arc_loader.js` requires jQuery.

How to define native functions (JS bridge)

You've written Hello world in Arc but there isn't a function named `js/log` yet. So you need to define a primitive function named `js/log` into ArcJS's namespace. Like this.

```
...
</body>
<script type="text/javascript">
```

```
var holder = $('#holder'), txt = '';
ArcJS.Primitives('user').define({
  'js/log': [{dot: -1}, function(log) {
    txt += log + "\n";
    holder.text(txt);
  }]
});
</script>
</html>
...
```

Then, `js/log` is defined into ArcJS's user namespace.

```
ArcJS.Primitives(string namespace).define({
  'name': [option, function]
});
```

As you see, you can define a native function into a specified namespace. `{dot: -1}` on `option` means that after the number of args will be a list have arbitrary length and will be passed like `:rest` parameter in CommonLisp. (`-1` means there is no rest parameters) For example, `(fn args...) => {dot: 0}` or `(fn a b args...) => {dot: 2}`.

The whole code will be like as follows.

```
<!doctype html>
<html>
  <head>
    <script type="text/javascript" src="//code.jquery.com/jquery-1.10.2.js"></script>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="arc_loader.js"></script>
    <script type="text/arc">
      (js/log "Hello world !!")
    </script>
  </head>
  <body>
    <textarea id="holder" style="width:500px;height:600px;"></textarea>
  </body>
  <script type="text/javascript">
    var holder = $('#holder'), txt = '';
    ArcJS.Primitives('user').define({
      'js/log': [{dot: -1}, function(log) {
        txt += log + "\n";
        holder.text(txt);
      }]
    });
  </script>
</html>
```

See example `hw.html`

How to call Arc functions from JavaScript

Then let's add FizzBuzz.

```
<script type="text/arc">

(js/log "Hello world !!")
```



```
(def FizzBuzz (l)
  (for n 1 l
    (js/log:string
      (case (gcd n 15)
        1 n
        3 'Fizz
        5 'Buzz
        'FizzBuzz))))

</script>
```

Then you have defined `FizzBuzz` function. You can call it from JavaScript like this:

```
<script type="text/javascript">
ArcJS.Primitives('user').define({ /* ... */ });
$(function(){
  var ctx = ArcJS.context();
  ctx.evaluate('(FizzBuzz 100)');
});
</script>
```

Or of course you can simply do:

```
<script type="text/arc">

(def FizzBuzz (l)
  ;; ...
  )

(FizzBuzz 100)

</script>
```

Whole code will be like as follows.

```
<!doctype html>
<html>
  <head>
    <script type="text/javascript" src="//code.jquery.com/jquery-1.10.2.js"></script>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="arc_loader.js"></script>
    <script type="text/arc">
      (js/log "Hello world !!")
      (def FizzBuzz (l)
        (for n 1 l
          (js/log:string
            (case (gcd n 15)
              1 n
              3 'Fizz
              5 'Buzz
              'FizzBuzz))))
    </script>
  </head>
  <body>
    <textarea id="holder" style="width:500px;height:600px;"></textarea>
  </body>
  <script type="text/javascript">
    var holder = $('#holder'), txt = '';
    ArcJS.Primitives('user').define({
```

```
'js/log': [{dot: -1}, function(log) {
  txt += log + "\n";
  holder.text(txt);
}]
});
$(function() {
  var ctx = ArcJS.context();
  ctx.evaluate('(FizzBuzz 100)');
});
</script>
</html>
```

See example `fizzbuzz.html`

More complicated example on website (Reversi player)

This is an automatic reversi player. The main search logic is written in Arc. It searches 3 turns depth by using depth-first search and Alpha-beta pruning. You can customize the depth and the space of searching by configuring `ev-depth` and `ev-space` in `reversi.arc`.

See example `reversi.html`

```
<!doctype html>
<html lang="en">
  <head>
    <title>Reversi</title>
    <script type="text/javascript" src="//code.jquery.com/jquery-1.10.2.js"></script>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="arc_loader.js"></script>
    <script type="text/arc" src="reversi.arc"></script>
  </head>
  <body>
    <h3>Reversi</h3>
    <canvas id="c1" width="600" height="600" style="float:left;border:1px solid #d3d3d3;"></canvas>
    <textarea id="holder" style="float:left;width:500px;height:600px;font-family:Consolas,Monaco,monospace"></textarea>
    <script type="text/javascript" src="reversi_bridge.js"></script>
    <script type="text/arc">
      (start-game)
    </script>
  </body>
</html>
```

The main search function in Arc.

```
(def get-best (board color depth)
  ((afn (board color depth target-color alpha beta)
    (if (or (is depth 0)
            (no (has-empty? board))) ;; last-depth or game-set
        (list (get-points board target-color)
              (withs (my-turn (is target-color color)
                          best-con (if my-turn > <)
                          best-fn (fn (a b) (best-con (car a) (car b)))
                          invert-color (invert color)))
              (iflet
                (puttable (get-puttable-positions-all board color)
                  (ccc
                    (fn (return)
                     (best
```

```

best-fn
  (map
    (fn (vp)
      (let new-board (put vp board color)
        (ret point-move (self new-board invert-color (- depth 1) target-color alpha)
          (let point (car point-move)
            (if my-turn
              (when (> point alpha)
                (= alpha point)
                (if (>= alpha beta)
                  (return (cons beta vp)))))) ;; alpha-cut
              (when (< point beta)
                (= beta point)
                (if (>= alpha beta)
                  (return (cons alpha vp)))))) ;; beta-cut
          (scdr point-move vp))))
      (get-rand puttable ev-space)))) ;; cut-off if candidates are over space.
    (self board invert-color (- depth 1) target-color alpha beta)))) ;; pass
board color depth color -inf.0 +inf.0))

```

See reversi.arc

And the bridge code.

```

(function() {
  var canvas = document.getElementById('c1'), size = 600;
  var txt = '', holder = $('#holder');

  function clear_board() {
    var ctx = canvas.getContext('2d');
    ctx.fillStyle = "rgb(0, 153, 0)";
    ctx.fillRect(0, 0, size, size);
    var unit = size / 8;
    for (var i = 1, l = 8; i<l; i++) {
      var x = i*unit;
      for (var j=0; j<2; j++) {
        ctx.beginPath();
        ctx.moveTo(j?0:x, j?x:0);
        ctx.lineTo(j?size:x, j?x:size);
        ctx.closePath();
        ctx.stroke();
      }
    }
  }

  function draw_stone(x, y, color) {
    var ctx = canvas.getContext('2d');
    var unit = size / 8;
    var xp = (unit*x) + (unit/2);
    var yp = (unit*y) + (unit/2);
    var r = unit * 0.85;
    ctx.beginPath();
    ctx.arc(xp, yp, r/2, 0, 2 * Math.PI, false);
    ctx.fillStyle = color;
    ctx.fill();
    ctx.lineWidth = 1;
    ctx.strokeStyle = '#000';
    ctx.stroke();
  }
}

```

```
function js_log() {
  txt += Array.prototype.slice.call(arguments).join(' ') + "\n";
  holder.text(txt);
  holder.scrollTop(holder[0].scrollHeight);
}

ArcJS.Primitives('user').define({
  'js/clear-board': [{dot:-1}, clear_board],
  'js/draw-stone':  [{dot:-1}, draw_stone],
  'js/log':         [{dot:0},  js_log]
});

clear_board();

})();

reversi_bridge.js
```

How to run on your machine

Run arc scripts

You can pre-compile arc code to JavaScript. Pass your scripts to the `arcjs` command.

```
$ echo "(prn (gcd 33 77))" > script.arc
$ arcjs script.arc
11
$
```

Run REPL with pre-loading arc scripts

Specify scripts after `-l` option.

```
$ echo "(def average (x y) (/ (+ x y) 2))" > avg.arc
$ arcjs -l avg.arc
arc> (average 10 20)
15
```

How to compile an arc script into JavaScript

Use `arcjsc` to compile arc to JavaScript.

```
$ echo "(def average (x y) (/ (+ x y) 2))" > avg.arc
$ arcjsc -o avg.js.fasl avg.arc
$ cat avg.js.fasl
// This is an auto generated file.
// Compiled from ['avg.arc'].
// DON'T EDIT !!!
preloads.push([
  [12,7,14,20,0,1,0,20,2,-1,0,10,9,1, ...
]);
preload_vals.push(["2","+","/", ...
$
```

avg.js.fasl

Then you can use the script in arcjs.

```
$ arcjs -l avg.js.fasl
arc> (average 10 20)
15
```

Or on a webpage.

```
<!doctype html>
<html lang="en">
  <head>
    <script type="text/javascript" src="//code.jquery.com/jquery-1.10.2.js"></script>
    <script type="text/javascript" src="arc.min.js"></script>
    <script type="text/javascript" src="arc_loader.js"></script>
    <script type="text/arc-fasl" src="avg.js.fasl"></script>
  </head>
  <body>
    <script type="text/arc">
      (prn (average 10 20)) ;; will be printed in console.log()
    </script>
  </body>
</html>
```

See example fasl_example.html

NameSpace reference

NOW WORK IN PROGRESS

arc.core

core utils for arc.

arc.arc

arc's main functions.

arc.collection

function set for collection.

arc.compiler

arc compiler functions.

arc.unit

for unit testing.

What is ArcJS?

ArcJS is an Arc-language implementation written in JavaScript. It consists of a self-hosting compiler written in Arc and a StackVM written in JavaScript.

Main features

- It runs on its StackVM ¹ so the Arc-code is compiled into asm-code in advance for performance.
- The Arc compiler is written in Arc itself. (Self-hosting compiler) ²
- macros, objects, first-class continuations and tail-call optimization are completely supported.
- Arc's characteristic features; default function, ssyntax are also supported.

ArcJS's original features (Not in official Arc-language)

- User defined ssyntax.
- clojure-like namespace system.
- new syntax (table, etc)
- stack tracer. (debugger)

¹ vm.js

² compiler.arc

What is Arc-language?

Arc-language is a dialect of the Lisp programming language now under development by Paul Graham and Robert Morris. Its simpler than Common-Lisp and more powerful than Scheme (IMO).¹

¹ Official Home Page