CLOJURESCRIPT.NET

TRANSLATIONS FROM JAVASCRIPT

Getting Started

Printing to the console

```
console.log("Hello, world!");
```

```
;; to print in browser console
(.log js/console "Hello, world!")

;; to print at ClojureScript REPL
(println "Hello, world!")
```

Code modularity

Define a library

```
// No native implementation
```

```
(ns my.library)
```

Use a library

```
// No native implementation
```

```
(ns my.library
  (:require [other.library :as other]))
```

Variables

Top Level

```
var foo = "bar";
```

```
(def foo "bar")
```

Local Variables

```
function foo() {
  var bar = 1;
}
```

```
(defn foo []
  (let [bar 1]))
```

Hoisting

```
// JavaScript "hoists" variables to the top of
// their scope.  So the following function:

function printName() {
  console.log('Hello, ' + name);
  var name = 'Bob';
}
```

```
;; ClojureScript does not hoist variables
;; this function will issue a warning

(defn print-name []
  (println "Hello, " name)
  (let [name "Bob"]))
```

```
// is equivalent to this function:

function printName() {
  var name;
  console.log('Hello, ' + name);
  name = 'Bob';
}

printName();
// Hello, undefined
```

Destructuring bind

```
// No native impelementation, must pull
// aparts objects and arrays manually

var o = {first: "Bob",
         middle: "J",
         last: "Smith"};

var first = o.first;
var last = o.last;
var middle = o.middle;
...

var color = [255, 255, 100, 0.5];
var red = color[0];
var green = color[1];
var alpha = color[3];
...
```

```
;; can always destructure in binding expression
;; including, let, function arguments, loops, etc.

(def m {:first "Bob"
        :middle "J"
        :last "Smith"})

(let [{:keys [first middle last]} m]
  ...)

(def color [255 255 100 0.5])

(let [[r g _ a] color]
  ...)
```

Dynamic binding

```
// can dynamic bind by putting
// object in scope chain
// performance implications

var x = 5;

var obj = {
  x: 10
};

with(obj) {
  console.log(x); // => 10
}

console.log(x); // => 5
```

```
;; efficient dynamic binding

(def ^:dynamic x 5)

(binding [x 10]
  (println x)) ;; => 10

(println x) ;; => 5
```

Mutable Locals

```
// In JavaScript locals are mutable

function foo(x) {
  x = "bar";
}
```

```
;; this will issue an error

(defn foo [x]
  (set! x "bar"))
```

Primitive Collections

Arrays

```
var a = new Array();
```

```
(def a (array))
```

```
var a = [];
var a = [1 2 3]
```

```
(def a (array 1 2 3))
```

Object

```
var o = {};
var o = new Object();
var o = {foo: 1, bar: 2};
```

```
(def o (js-obj))
(def o (js-obj "foo" 1 "bar" 2))
```

Immutable Collections

Immutable Lists

```
// No native implementation
```

```
;; efficient addition at head

(def l (list))
(def l (list 1 2 3))
(def l '(1 2 3))
(conj l 4) ;; => '(4 1 2 3)
```

Immutable Vectors

```
// No native implementation
```

```
;; efficient addition at the end

(def v (vector))
(def v [])
(def v [1 2 3])
(conj v 4) ;; => [1 2 3 4]
```

Immutable Sets (Collection of unique items)

```
// No native implementation
```

```
(def s (set))
(def s #{})
(def s #{"cat" "bird" "dog"})
(conj s "cat") ;; => #{"cat" "bird" "dog"}
```

Immutable HashMaps

```
// No native implementation
```

```
(def m (hash-map))
(def m {})
(def m {:foo 1 :bar 2})
(conj m [:baz 3]) ;; => {:foo 1 :bar 2 :baz 3}
```

Accessing Values

```
// map access is a static
// language feature

var m = {
  "foo": 1,
  "bar": 2
};
```

```
// collection access is first class

(def m {:foo 1
        :bar 2})

(get m :foo)
```

```
m["foo"];
m.foo;

// array access is a static
// language feature

var a = ["red", "blue", "green"];
a[0];
```

```
(def v ["red" "blue" "green"])

(nth v 0)
```

Arbitrary Keys

```
// Only string keys allowed

var m = {
  "foo": 1,
  "bar": 2
};
```

```
;; Arbitrary keys allowed

(def m { [1 2] 3
         #{1 2} 3
         '(1 2) 3 })
```

Adding to a collection

```
var a = [];
a.push("foo"); // destructive update

var b = {};
b["bar"] = 1; // destructive update
```

```
(def a [])
(conj a "foo") ;; => ["foo"]
;; efficient non-destructive update

(def b {})
(assoc b :bar 1) ;; => {:bar 1}
;; efficient non-destructive update
```

Callable Collections

```
// No native implementation
// array and object access
// is a static language feature
```

```
(def address {:street "1 Bit Ave."
              :city "Bit City"
              :zip 10111011})

(map address [:zip :street])
;; => (10111011 "1 Bit Ave.")
;; Collections can act as
;; functions. HashMaps are functions
;; of their keys.
```

Cloning

```
var a = [...];
foo(a.slice(0));

// if foo might mutate a, must clone,
// however this is only a shallow copy
// no native implementation of deep copy
// for primitive collections
```

```
(def a [...])
(foo a)

;; shallow cloning, deep cloning unnecessary
;; all core collections are immutable
```

Equality

```
// No native implementation

var a = ["red", "blue", "green"];
var b = ["red", "blue", "green"];
```

```
(def a ["red" "blue" "green"])
(def b ["red" "blue" "green"])
(= a b) ;; => true
```

```
console.log(a == b); // => false

// == tests identity
// must implement your own deep
// equality test for all types
```

Booleans

If Statements

```
var bugNumbers = [3234,4542,944,124];

if (bugNumbers.length > 0) {
  console.log('Not ready for release');
}
```

```
(def bug-numbers [3234 452 944 124])

(if (pos? (count bug-numbers))
  (println "Not ready for release"))
```

Handling of empty strings

```
var emptyString = "";

if (emptyString) {
  console.log("You won't see me!");
} else {
  console.log("Empty string is treated"+
              " as false!");
}
```

```
(def empty-string "")

(if empty-string
  (println "Empty string is not false!"))
```

Handling of zero

```
var zero = 0;

if (zero) {
  console.log("You won't see me!");
} else {
  console.log("Zero is treated as false!");
}
```

```
(def zero 0)

(if zero
  (println "Zero is not false!"))
```

Value and Identity Equality

```
// == operator is coercive
1 == "1" // => true

// sometimes based on value
{} == {} // => false
```

```
;; ClojureScript has no coercive
;; equality operator, equality
;; is always based on value

(= 1 "1") ;; => false
(= {} {}) ;; => true (see Collections)
```

Functions

Function definition

```
function foo() {
  ...
  return true;
}
```

```
(defn foo []
  true)
```

Return Value

```
// JavaScript is statement oriented
// explicit return

function foo() {
  ...
  return true;
}
```

```
;; ClojureScript is expression oriented
;; no explicit return

(defn foo []
  true)
```

Assign a function to a variable

```
var foo = function() {
  ...
  return true;
}
```

```
(def foo (fn [] true))
```

Optional Parameters

```
function foo(a, b, c) { return c; };

foo(1) // => undefined
foo(1, 2, 3) // => 3
```

```
(defn foo [a b c] c)

(foo 1) ;; WARNING: function called with incorrect
        ;; number of arguments

(foo 1 2 3) ;; => 3
```

Dispatch on arity

```
// No native implementation must manipulate
// arguments object - performance implications
```

```
(defn foo
  ([a] "one")
  ([a b] "two")
  ([a b c] "three"))

(foo 1) ;; => "one"
(foo 1 2) ;; => "two"
(foo 1 2 3) ;; => "three"

;; Under advanced compilation direct dispatch to
;; arity. No arguments object manipulation
```

Variable arguments

```
// No native implementation. Manipulate arguments
// object explicitly. Performance implications.

function foo() {
  var args = arguments;
  ...
}
```

```
;; all arguments beyond two will be placed in a
;; sequence bound to rest

(defn foo [a b & rest]
  ...)
```

Named Parameters & Defaults

```
// No native implementation. Pass objects
```

```
(defn foo [& {:keys [bar baz]}]
```

```
// explicitly.

function foo(o) {
  var bar = o.bar,
      baz = o.baz;
  ...
}

foo({bar: 1, baz: 2});

function foo(o) {
  var bar = o.bar || "default1",
      baz = o.baz || "default2";
  ...
}
```

```
  ...)

(foo :bar 1 :baz 2)

(defn foo [& {:keys [bar baz]
              :or {bar "default1"
                   baz "default2"}}]
  ...)
```

Iterators

Uniform Iteration For All Types

```
// JavaScript does not have uniform iteration
// over native types.

var colors = ['red', 'orange', 'green'];

for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}

var data = { ... };

for (var key in data) {
  console.log('key', key);
  console.log('value', data[key]);
}
```

```
;; All ClojureScript datastructures support
;; uniform iterations. JavaScript natives are
;; safely extended to the same iteration protocol

(def colors (array "red" "orange" "green"))

(doseq [color colors]
  (println color))

(def colorsv ["red" "orange" "green"])

(doseq [color colorsv]
  (println color))

(def data { ... })

(doseq [[k v] data]
  (println "key" k)
  (println "value" v))
```

Closure and counters in loops

```
var callbacks = [];

// A closure must be used to preserve the
// return for each function at each step of
// the loop.  Otherwise every entry in
// callbacks will return 2;

for (var i = 0; i < 2; i++) {
  (function(_i) {
    callbacks.push(function() {
      return _i;
    });
  })(i);
}

// Without the internal closure,
// the result is 2

callbacks[0]() // == 0

// ECMAScript 6 can support this with
// the use of blocks

let callbacks = [];
for (let i = 0; i < 10; i++) {
  let j = i;
  callbacks.push(function() { print(j) });
}
```

```
;; ClojureScript has proper lexical scope

(def callbacks (atom []))

(dotimes [i 2]
  (swap! callbacks conj (fn [] i)))

((@callbacks 0)) // => 0
```

Lazy Sequences

Map

```
// this will traverse two arrays

var colors = ["red", "green", "blue"];
console.log((colors.map(function(s) {
  return s[0];
})).map(function(c) {
  return c + "foo";
})); // => ["rfoo", "gfoo", "bfoo"]
```

```
;; lazy, will only traverse once

(def colors ["red" "green" "blue"])

(println
  (map #(str % "foo") (map first colors))
;; => ("rfoo" "gfoo" "bfoo")
```

Filter

```
var numbers = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10];

var filtered = numbers.filter(function(n) {
  return n % 5 == 0;
}); // filters entire array

var firstn = filtered[0];
// but we only want the first one
```

```
(def numbers [0 1 2 3 4 5 6 7 8 9 10])

(def filtered
  (filter #(zero? (rem % 5)) numbers))

(def firstn (first filtered))

;; lazy filter, values after 5 haven't
;; been looked at
```

Types

Define

```
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  return "Hello, " + this.name;
}
```

```
(deftype Person [name]
  Object
  (greet [_]
    (str "Hello" name)))

;; Constructors don't look like functions
;; No explicit prototype manipulation
;; No explicit 'this' to access fields
```

Instantiate

```
var person = new Person("Bob");
```

```
(Person. "Bob")
```

Reflection

```
var name = "Bob";
typeof name // => "string"
```

```
;; reflection returns constructors not strings

(def name "Bob")
(type name) ;; => String
```

Check Type

```
var name = "Bob";

name instanceof String // => true
```

```
(def name "Bob")

(= (type name) js/String) ;; => true
```

```
(!(name instanceof Number)) // => true
```

```
(string? name) ;; => true

(not= (type name) js/Number) ;; => true
(not (number? name)) ;; => true
```

Protocols

```
// Duck typing, implement two different types
// with the same method names

function Cat() {};
Cat.prototype.sound = function() {
  return "Meow!";
}

function Dog() {};
Dog.prototype.sound = function() {
  return "Woof!";
}

// lacking indirection no way to provide defaults
(1).sound() // Error
```

```
(defprotocol ISound (sound []))

(deftype Cat []
  ISound
  (sound [_] "Meow!"))

(deftype Dog []
  ISound
  (sound [_] "Woof!"))

(extend-type default
  ISound
  (sound [_] "... silence ..."))

(sound 1) ;; => "... silence ..."
```

Regular Expressions

```
var email = "test@example.com";
email.match(/@/)
// => ["@"]
```

```
(def email "test@example.com")
(.match email #"@")
;; => ["@"]
```

```
var invalidEmail = "f@il@example.com";
invalidEmail.match(/@/g)
// => ["@", "@"]
```

```
(def invalid-email "f@il@example.com")
(re-seq #"@" invalid-email)
;; => ("@" "@")
```

Exceptions

Throw an exception

```
throw Error("Oops!");
```

```
(throw (js/Error. "Oops!"))
```

Catch an exception

```
try {
  undefinedFunction();
} catch(e) {
  if (e instanceof ReferenceError) {
    console.log('You called a function'+
                'that does not exist');
  }
} finally {
  console.log('This runs even if'+
              'an exception is thrown');
}
```

```
(try
  (undefined-function)
  (catch js/Error e
    (if (= (type e) js/ReferenceError)
      (println
        (str "You called a function"
             "that does not exist"))))
  (finally
    (println
      (str "this runs even if an"
           "exception is thrown"))))
```

Expression Problem

Modifying Types You Don't Control

```
// JavaScript allows you to modify prototypes

String.prototype.foo = function(...) {
  ...
};

// Because of the likelihood of clashes this is
// considered bad practice
```

```
;; ClojureScript namespaces everything, you
;; can make local extensions with little worry

(defprotocol MyStuff
  (foo [this]))

(extend-type string
  MyStuff
  (foo [this]
    ...))

;; In addition native JavaScript objects like
;; Function, Object, Array, Number, String
;; are never actually directly extended

;; For example say you'd like to use RegExps
;; as functions

(extend-type js/RegExp
  IFn
  (-invoke
   ([this s]
     (re-matches this s))))

(filter #"foo.*" ["foo" "bar" "foobar"])
;; => ("foo" "foobar")
;; This is precisely how callable collections
;; are implemented.
```

Metaprogramming

Runtime

```
// JavaScript is dynamic, standard runtime
// metaprogramming techniques applicable
```

```
;; ClojureScript is dynamic, standard runtime
;; metaprogramming techniques applicable
```

Compile Time

```
// No native implementation, must use external
// compilation tools.
```

```
;; ClojureScript has compiler macros, no external
;; tool required

(defmacro my-code-transformation [...]
  ...)

;; Ocaml, Haskell style pattern matching is a
;; library.

;; Prolog style relational programming is a
;; library
```