Hot Cocoa Lisp

Douglas Crockford has called JavaScript "Lisp in C's clothing"[1]. Hot Cocoa Lisp could be described as an attempt to put it back into Lisp's clothing. The basic idea is a language with the syntax of Lisp that compiles to JavaScript, much like CoffeeScript. Like CoffeeScript, you should be able to use existing JavaScript libraries seamlessly from Hot Cocoa Lisp and vice-versa. Hot Cocoa Lisp emphasizes functional programming by providing a natural syntax for highly nested function calls and providing tools to make common functional idioms more natural while maintaining all of the good parts of JavaScript.

This implementation runs using Node.js as a convenient command line interface for JavaScript. It is often convenient to also use Node.js to run code compiled from Hot Cocoa Lisp. Many of the examples below assume basic familiarity with the Node.js CLI.

Basic Syntax

The syntax of Hot Cocoa Lisp is a combination of that of JavaScript and that of Lisp. Functions calls take the form of Lisp style S-expressions and, like Lisp, every syntax in the language takes the form of function call. Additionally, the dotted object access syntax from JavaScript (e.g. `foo.bar`) is allowed. So for example:

Hot Cocoa Lisp

```
(console.log "Hello World!")
```

JavaScript

```
console.log("Hello World!");
```

Getting Started

Hot Cocoa Lisp can be installed using npm:

```
$ npm -g install hot-cocoa-lisp
```

Once installed a file can be compiled and run using `hcl -n`:

```
$ cat hello.hcl
(console.log "Hello World!")

$ hcl -n hello.hcl
Hello World!
```

There is also a REPL:

```
$ hcl
hcl> (+ 1 2 3)
6
```

More instructions can be found using `hcl --help`

Arrays, Objects, and Variables

Array and object literals can be declared exactly as they can in JavaScript with the exception that the commas and colons are optional.

Elements of arrays and objects can be accessed using `get`. (Or synonymously `nth`). `get` is equivalent to the JavaScript syntax `obj[key]`.

Variables can be initialized with `var` or `def`. The two are synonyms but `def` is preferred when the value is expected not to change and `var` is preferred when it might change.

The values of variables and elements of objects and arrays can be modified using `set` which is equivalent to the JavaScript `a = b`. `set` has a special form with 3 arguments which can be used to modify an element of an object or array and is equivalent to the JavaScript `a[b] = c`.

Hot Cocoa Lisp

```
;; initialize variables
(var foo 7)
(var bar [ 1 2 3 ] )
(var baz)

;; access
(console.log (get bar 0)) ; 1

;; assignments
(set (get bar 3) 4)
(set baz { one 1 two 2 } )
(set baz.three 3)
(set (get baz "four") 4)
(set baz "four" 4) ; alternate special form
```

JavaScript

```
var foo = 7, bar = [1, 2, 3], baz;
```

```
console.log(bar[0]); // 1

bar[3] = 4;
baz = { one: 1, two: 2 };
baz.three = 3;
baz["four"] = 4;
```

It should be noted that unlike Lisp, words in places besides the beginning of an S-expression are not symbols that can be manipulated as values, evaluated, or quoted to prevent evaluation. Rather they are simply identifiers and they work exactly as they would in JavaScript with the exception that they aren't limited to the characters normally allowed in JavaScript identifiers (see Syntax Details below).

As with Lisp, the `;` character begins a one line comment.

Functions and Loops

functions

Function literals (a.k.a. lambdas) can be defined using `lambda`, which can also be spelt `function`, or the much shorter `#`. Because in JavaScript (just like Scheme) functions occupy the same namespace as other variables, a simple function definition can be made using a combination of `def` and `#`:

Hot Cocoa Lisp

```
(def factorial
    (# (n)
        (if (< n 2) 1
            (* n (factorial (- n 1))))))
```

JavaScript

```
var factorial = (function(n) { return ((n < 2) ? 1 : factorial(n - 1)) });
```

Three basic loops are provided in Hot Cocoa Lisp: `while`, `times`, and `for`.

while

`while` is the familiar while loop from JavaScript:

Hot Cocoa Lisp

```
(var input)
(while (!= "exit" (set input (prompt "enter a command")))
    (alert (cat "You entered the command: " input)))
(alert "Goodbye")
```

JavaScript

```
var input;
(function() {while ((("exit" !== (input = prompt("enter a command"))))) { alert(("You entere
alert("Goodbye");
```

It's worth noting that the normal JavaScript looping construct gets wrapped in a function call here to ensure that while expression can be nested inside of larger expression for example:

Hot Cocoa Lisp

```
(var input)
(if (!= "yes" (prompt "do you want to enter a loop?")) (alert "ok")
    (while (!= "exit" (set input (prompt "enter a command")))
        (alert (cat "You entered the command: " input))))
(alert "Goodbye")
```

JavaScript

```
var input;
((!== "yes" prompt("do you want to enter a loop?")) ? alert("ok") : (function() {while ((("e
alert("Goodbye");
```

times

`times` is a standard loop that counts up from 0 to n:

Hot Cocoa Lisp

```
(times (x 10)
    (console.log x))
```

JavaScript

```
(function() { for (var x = 0; x < 10; x++) { console.log(x); }}).call(this);
```

for

`for` takes two forms depending on whether its first argument has 2 or 3 elements. With 2 the `for` acts like a Python-style for loop. With 3 the `for` acts like a for loop from JavaScript or C.

Hot Cocoa Lisp

```
(for (x [ 2 4 6 ] )
    (console.log(x)))

(for ((var x 1) (< x 100) (set* x 2))
    (console.log(x)))
```

JavaScript

```
(function() { for (var _i_ = 0; _i_ < [2, 4, 6].length; _i_++) { var x = [2, 4, 6][_i_]; co

(function() { for (var x = 1; x < 100; x *= 2) { console.log(x); }}).call(this);
```

The `set*` above is equivalent to the JavaScript `*=` operator. There are similar constructs for other arithmetic operators including `set+`, `set-`, and `set/`.

Splats

As with JavaScript, user defined functions can take any number of arguments. If more arguments are specified in the API then are passed in the call, the extra arguments are assigned the value `undefined`. All arguments are then stored in the `arguments` object. Unlike JavaScript, arbitrary arguments can also be captured using splats. If the last argument specified in the API ends with three dots then all remaining arguments passed are stored in an array.

Hot Cocoa Lisp

```
(def print-numbers
    (# (label numbers...)
        (for (n numbers)
            (console.log (cat label ": " n)))))

(print-numbers "num" 1 2 3)
```

JavaScript

```
print_hyphen_numbers = (function(label) {var numbers = [].slice.call(arguments, 1), _i0_, n

print_hyphen_numbers("num", 1, 2, 3);
```

Output

```
num: 1
num: 2
num: 3
```

Annotations

One of the downsides of emphasizing a functional style is that making the compiled JavaScript Source readable is basically a lost cause. For this reason inline annotations of the original source code are added to the compiled output to provide context for debugging purposes.

Hot Cocoa Lisp

```
;; define a function
(def do-math (# (x) (+ (* 7 x) (- x / 3))))

;; complex functional expression
(times (x 10)
    (if (< x 5)
        (console.log (do-math x))
      (console.log (do-math (do-math x)))))
```

JavaScript

```
var do_hyphen_math, x;

// ;; define a function
// (def do-math (# (x) (+ (* 7 x) (/ x 3))))

do_hyphen_math = (function(x) {  return ((7 * x) + (x / 3)); });

// ;; complex functional expression
// (times (x 10)
//     (if (< x 5)
//         (console.log (do-math x))
//       (console.log (do-math (do-math x)))))

(function() {for (x = 0; x < 10; x++) { (((x < 5)) ? console.log(do_hyphen_math(x)) : conso
```

Underscore

The -u flag makes Underscore.js 1.4.3 available at the top level. Each property of the _ object added as a top level function. For example:

```
(def _ (require "underscore"))

(console.log (_.map [ 1 2 3 ] (# (x) (+ 2 x))))
```

could be written using the -u flag as

```
(console.log (map [ 1 2 3 ] (# (x) (+ 2 x))))
```

This also works in the REPL.

Syntax Details

boolean literals

The literal values `true`, `false`, `null`, `undefined`, and `NaN` work exactly as they do in JavaScript with the exception that the built-in functions for determining the types of values will differentiate `null` and `NaN` into their own types (null and nan respectively).

```
(type true) ; "boolean"
(type false) ; "boolean"
(type null) ; "null"
(type undefined) ; "undefined"
(type NaN) ; "nan"
```

number literals

Number literals follow the JSON specification[2] for numbers. That is to say anything that matches the regular expression `-?(0|[1-9][0-9]*)(\.[0-9]+)?([eE][-+]?[0-9]+)?` is parsed as a number. Additionally `Infinity` and `-Infinity` are interpreted as numbers.

string literals

String literals work just as they do in JavaScript.

identifiers

Identifiers in may contain any combination of letters digits or any of the following characters:

`_`, `!`, `?`, `$`, `%`, `&`, `@`, `#`, `|`, `~`, `*`, `+`, `-`, `=`, `/`, `<`, `>`, `^`, or ‘

Identifiers may not begin with digits or be interpretable as a number. For example `-1` will be parsed as a number and not an identifier. Identifiers with characters that aren't normally allowed in JavaScript are represented by replacing all occurrences of symbols not normally allowed in JavaScript identifiers with underscore delimited place-holders, for example `a/b` becomes `a_slash_b`. To prevent accidental overlap `_` is replaced with `__`. This creates an inconvenience in the case that global variables from external libraries written in JavaScript contain underscores. It is always possible to access them via the global object (`global` in Node.js or `window` in a browser) however because this can be somewhat unreliable, the `from-js` function is provided:

External JavaScript Library

```
some_cool_global_variable = { ... }
```

Hot Cocoa Lisp

```
(make-use-of (from-js some_cool_global_variable))
```

Compiled JavaScript

```
make_hyphen_use_hyphen_of(some_cool_global_variable);
```

array and object literals

Array and object literals are translated by the parser into S-expressions so:

```
{ colors [ "red" "blue" "green" ]
  shapes [ "square" "triangle" ] }
```

is equivalent to

```
(object colors (array "red" "blue" "green")
        shapes (array "square" "triangle"))
```

either will be compiled to

```
{ colors: ["red", "blue", "green"], shapes: ["square", "triangle"] }
```

dotted object access

Dotted object access is also translated by the parser to S-expressions. This means that

```
(set foo.bar (snap.crackle.pop))
```

is equivalent to

```
(set (. foo bar) ((. snap crackle pop)))
```

which will be compiled to

```
foo.bar = snap.crackle.pop();
```

Hot Cocoa Lisp

```
;; define a function
(def do-math (# (x) (+ (* 7 x) (- x / 3))))

;; complex functional expression
(times (x 10)
    (if (< x 5)
        (console.log (do-math x))
      (console.log (do-math (do-math x)))))
```

JavaScript

```
var do_hyphen_math, x;

// ;; define a function
// (def do-math (# (x) (+ (* 7 x) (/ x 3))))

do_hyphen_math = (function(x) {  return ((7 * x) + (x / 3)); });

// ;; complex functional expression
// (times (x 10)
//     (if (< x 5)
//         (console.log (do-math x))
//       (console.log (do-math (do-math x)))))

(function() {for (x = 0; x < 10; x++) { (((x < 5)) ? console.log(do_hyphen_math(x)) : consol
```

Function reference

The following reference describes all of the syntaxes built into Hot Cocoa Lisp.

Functions marked with * don't necessarily compile all of their arguments.

Functions marked with ** can be accessed by their names within the language.
For example:

```
;; this should be compiled with the -u flag to include map from underscore
(map [ 1 2 3 ] +1) ; [ 2 3 4 ]
```

---

nop

**

```
 (nop)
```

Takes 0 arguments and returns `undefined`.

Hot Cocoa Lisp

```
(def bogus-function (# () (nop)))
```

JavaScript

```
var bogus_hyphen_function = (function() { return undefined; });
```

.

*

```
(. object keys...)
```

Takes 2 or more arguments and does chained object access.

Hot Cocoa Lisp

```
(. object key1 key2)
```

JavaScript

```
object.key1.key2;
```

---

get

**

Synonyms: `nth`

```
(get object key)
```

Takes 2 arguments and does object or array access.

Hot Cocoa Lisp

```
(get object key)
```

JavaScript

```
object[key]
```

---

list

Synonyms: `array`

```
(list ...)
```

Takes 0 or more arguments and creates an array literal.

Hot Cocoa Lisp

```
(list 1 2 3 4)
```

JavaScript

```
[1, 2, 3, 4]
```

---

object

*

```
(object ...)
```

Takes an even number of arguments and creates an object literal. The first in each pair of arguments is interpreted as a key (and converted to a string if it is an identifier) and the second in each pair is compiled and interpreted as a value.

Hot Cocoa Lisp

```
(object x 1 y 2 z 3)
```

JavaScript

```
{ "x": 1, "y": 2, "z": 3 }
```

---

inherit

**

Synonyms: `new`

```
(inherit obj)
```

Takes 1 argument and creates a new object that inherits prototypally from it.

Hot Cocoa Lisp

```
(var foo { a 1 } )
(var bar (inherit foo))
(get bar "a") ; 1
```

JavaScript

```
var foo = { a: 1 };
var bar = Object.create(foo);
bar["a"] ; 1
```

---

if

**

```
 (if condition yes no)
```

Takes 3 arguments. Evaluates and returns the second if the first evaluates to true. Otherwise evaluates and returns the third.

Hot Cocoa Lisp

```
(if (= 1 number) "one" "not one")
```

JavaScript

```
((1 === number) ? "one" : "not one")
```

---

begin

```
(begin statements...)
```

Takes 1 or more arguments, evaluates each of them in turn and returns the last.

Hot Cocoa Lisp

```
(if (condition)
    (begin (thing1) (thing2))
  (begin (thing3) (thing4)))
```

JavaScript

```
(condition() ? (function() { thing1(); return thing2(); }).call(this) : (function() { thing3
```

---

when

```
 (when condition statements...)
```

Takes 2 or more arguments. Check whether the first evaluates to true then evaluates the rest if the first was true. Returns the last statement evaluated.

Hot Cocoa Lisp

```
(when (output-requested)
    (console.log things)
    (document.write things))
```

JavaScript

```
(output_hyphen_requested() && (function() { console.log(things); return document.write(thing
```

---

cond

*

```
(cond conditions...)
```

Takes 1 or more condition/result pairs. Evaluates and returns the first result whose condition returns true and `undefined` if none do.

Hot Cocoa Lisp

```
(cond
    ((list? x) (x.join " "))
    ((object? x) "[OBJECT]")
    (true x))
```

JavaScript

```
((Object.prototype.toString.call(x) === "[object Array]") ? x.join(" ") : (x !== null && typ
```

---

while

```
 (while condition statements...)
```

Takes 2 or more arguments. Evaluates the arguments after the first repeatedly as long as the first evaluates to true. Returns `undefined`.

Hot Cocoa Lisp

```
(var i 10)
(while (-- i)
    (alert i))
```

JavaScript

```
var i = 10;
(function() {while (i--) { alert(i); }}).call(this);
```

---

for

*

```
(for loop_init statements...)
```

Takes 2 or more arguments.

If the first argument is has two elements:

```
(for (iterator array) statements...)
```

The statements are evaluated once for each element of the array with the iterator assigned to that element.

If the first argument is has three elements:

```
(for (init condition iterator) statements...)
```

The init is evaluated once at the beginning then until the condition evaluates to false the statements are evaluated and then the iterator is evaluated.

Returns `undefined`.

Hot Cocoa Lisp

```
(for (x my-list)
    (alert x))

(for ((var x 1) (< x 20) (set* x 2))
    (alert x))
```

JavaScript

```
(function() {for (var _i0_ = 0; _i0_ < my_hyphen_list.length; _i0_++) { var x = my_hyphen_li

(function() {for (var x = 1; ((x < 20)); (x *= 2)) {  alert(x); }}).call(this);
```

---

times

*

`(times (iterator number) statements...)`

Takes 2 or more arguments. The statements are evaluated once for each non-negative integer less than the number with the iterator assigned to that integer. Returns `undefined`.

Hot Cocoa Lisp

```
(times (_ 10) (process.stdout.write " "))
```

JavaScript

```
(function() {for (__ = 0; __ < 10; __++) { process.stdout.write(" "); }}).call(this);
```

---

error

Synonyms: `throw`

`(error message)`

Takes 1 argument. Throws a new error with the specified message.

Hot Cocoa Lisp

```
(error "This isn't meant to happen")
```

JavaScript

```
(function() {throw new Error("This isn't meant to happen");}).call(this);
```

---

attempt

*

```
(attempt try [catch] [finally])
```

Takes 2-3 arguments. Creates a try..catch..finally block. It can take any of the following three forms:

Hot Cocoa Lisp

```
(attempt
    (try (things))
    (catch e
        (handle-error e)))
```

JavaScript

```
try {
    things();
} catch (e) {
    handle_hyphen_error(e);
}
```

Hot Cocoa Lisp

```
(attempt
    (try (things))
    (finally    (finish things)))
```

JavaScript

```
try {
    things();
} finally {
    finish(things);
}
```

Hot Cocoa Lisp

```
(attempt
    (try (things))
    (catch e
        (handle-error e))
    (finally (finish things)))
```

JavaScript

```
try {
    things();
} catch (e) {
    handle_hyphen_error(e);
} finally {
    finish(things);
}
```

Returns `undefined`.

+

**

Synonyms: `cat`

```
(+ summand1 additional_summands...)
```

Takes 2 or more arguments and applies the JavaScript `+` operator to them in order.

Hot Cocoa Lisp

```
(+ 1 2 3) ; 6
(cat "foo" "bar" "baz") ; "foobarbaz"
```

JavaScript

```
(1 + 2 + 3) // 6
("foo" + "bar" + "baz") // "foobarbaz"
```

+1

**

```
 (+1 summand)
```

Takes 1 argument. Adds 1 to the summand.

Hot Cocoa Lisp

```
(+1 7) ; 8
```

JavaScript

```
(7 + 1) // 8
```

-

**

```
 (- args...)
```

Takes 1 or more arguments.

Take one of the following two forms:

```
(- number)
```

The opposite of the number is returned.

```
(- minuend subtrahends...)
```

Subtracts each of the subtrahends from the minuend.

Hot Cocoa Lisp

```
(- 7) ; -7
(- 7 2) ; 5
(- 7 2 3) ; 2
```

JavaScript

```
(- 7) // -7
(7 - 2) // 5
(7 - 2 - 3) // 2
```

–1

**

```
 (--1 minuend)
```

Takes 1 argument. Subtracts 1 from the minuend.

Hot Cocoa Lisp

```
(--1 8) ; 7
```

JavaScript

```
(8 - 1) // 7
```

*****

**

```
 (* factor1 additional_factors...)
```

Takes 2 or more arguments and multiplies them together.

Hot Cocoa Lisp

```
(* 2 3 4) ; 24
```

JavaScript

```
(2 * 3 * 4) // 24
```

***2**

**

Synonyms: `double`

```
(*2 factor)
```

Takes 1 argument. Multiplies the factor by 2.

Hot Cocoa Lisp

```
(*2 5) ; 10
```

JavaScript

```
(5 * 2) // 10
```

/

**

```
 (/ dividend divisors...)
```

Takes 2 or more arguments. Divides the dividend by each of the divisors.

Hot Cocoa Lisp

```
(/ 10 2) ; 5
(/ 100 2 5) ; 10
```

JavaScript

```
(10 / 2) // 5
(100 / 2 / 5) // 10
```

/2

**

Synonyms: `half`

```
(/2 dividend)
```

Takes 1 argument. Divides the dividend by 2.

Hot Cocoa Lisp

```
(/2 8) ; 4
```

JavaScript

```
(8 / 2) // 4
```

---

^

**

```
 (^ base exponent)
```

Takes 2 arguments. Returns the base to the power of the exponent.

Hot Cocoa Lisp

```
(^ 2 5) ; 32
```

JavaScript

```
Math.pow(2, 5) // 32
```

---

^2

**

Synonyms: `square`

```
(^2 base)
```

Takes 1 argument. Squares the base.

Hot Cocoa Lisp

```
(^2 9) ; 81
```

JavaScript

```
(9 * 9) // 81
```

---

sqrt

**

```
 (sqrt number)
```

Takes 1 argument. Returns the square root of the number.

Hot Cocoa Lisp

```
(sqrt 100) ; 10
```

JavaScript

```
Math.sqrt(100) // 10
```

%

**

Synonyms: `mod`

```
(% number moduli...)
```

Takes 2 or more arguments. For each of the moduli the number is replaced by the remainder of number / modulus.

Hot Cocoa Lisp

```
(% 26 10) ; 6
(% 1010 100 7) ; 3
```

JavaScript

```
(26 % 10) // 6
(1010 % 100 % 7) // 3
```

<

**

Synonyms: `lt?`

```
(< number additional_numbers...)
```

Takes 2 or more arguments. Returns true if the each number is larger than the previous.

Hot Cocoa Lisp

```
(< 2 3)
(< 10 15 20 25)
```

JavaScript

```
(2 < 3)
((10 < 15) && (15 < 20) && (20 < 25))
```

>

**

Synonyms: `gt?`

```
(> number additional_numbers...)
```

Takes 2 or more arguments. Returns true if the each number is smaller than the previous.

Hot Cocoa Lisp

```
(> 3 2)
(> 25 20 15 10)
```

JavaScript

```
(3 > 2)
((25 > 20) && (20 > 15) && (15 > 10))
```

<=

**

Synonyms: `lte?`

```
(<= number additional_numbers...)
```

Takes 2 or more arguments. Returns true if the each number is smaller than or equal to the previous.

Hot Cocoa Lisp

```
(<= 2 3)
(<= 15 15 20 25)
```

JavaScript

```
(2 <= 3)
((15 <= 15) && (15 <= 20) && (20 <= 25))
```

>=

**

Synonyms: `gte?`

```
(>= number additional_numbers...)
```

Takes 2 or more arguments. Returns true if the each number is larger than or equal to the previous.

Hot Cocoa Lisp

```
(>= 3 2)
(>= 20 20 15 10)
```

JavaScript

```
(3 >= 2)
((20 >= 20) && (20 >= 15) && (15 >= 10))
```

=

**

Synonyms: `is`, `is?`, `eq`, `eq?`, `equal`, `equal?`, `equals`, `equals?`

```
(= arg1 args...)
```

Takes 2 or more arguments. Returns true if all of the arguments are equal.

Hot Cocoa Lisp

```
(= 2 2)
(= 20 20 20)
```

JavaScript

```
(2 === 2)
((20 === 20) && (20 === 20))
```

!=

**

Synonyms: `isnt`, `isnt?`, `neq`, `neq?`

```
(!= arg1 args...)
```

Takes 2 or more arguments. Returns true if none of the adjacent arguments are equal.

Hot Cocoa Lisp

```
(= 2 3)
(= 20 21 22)
```

JavaScript

```
(2 !== 3)
((20 !== 21) && (21 !== 22))
```

=0

**

Synonyms: `zero?`

```
(=0 number)
```

Takes 1 argument. Returns true if the number is 0.

Hot Cocoa Lisp

```
(=0 0)
```

JavaScript

```
(0 === 0)
```

&

**

Synonyms: `bit-and`

```
(& number1 number2)
```

Takes 2 arguments and performs a bitwise and between them.

Hot Cocoa Lisp

```
(& 6 3) ; 7
```

JavaScript

```
(6 & 3) // 7
```

`|`

Synonyms: `bit-or`

**

```
(| number1 number2)
```

Takes 2 arguments and performs a bitwise or between them.

Hot Cocoa Lisp

```
(| 6 3) ; 2
```

JavaScript

```
(6 | 3) // 2
```

<<

**

Synonyms: `bit-shift-left`

```
(<< number bits)
```

Takes 2 arguments. Bit-shifts the number to the left by the specified number of bits.

Hot Cocoa Lisp

```
(<< 1 2) ; 4
```

JavaScript

```
(1 << 2) // 4
```

>>

**

Synonyms: `bit-shift-right`

```
(>> number bits)
```

Takes 2 arguments. Bit-shifts the number to the left by the specified number of bits.

Hot Cocoa Lisp

```
(>> 4 2) ; 1
```

JavaScript

```
(4 >> 2) // 1
```

---

not

**

Synonyms: `not?`, `!`

```
(not arg)
```

Takes 1 argument and returns the boolean opposite of it.

Hot Cocoa Lisp

```
(not true) ; false
```

JavaScript

```
(! true) ; false
```

---

and

**

Synonyms: `and?`, `&&`

```
(and arg1 arg2...)
```

Takes 2 or more arguments and returns true if all of them evaluate to true.

Hot Cocoa Lisp

```
(and true true true)
```

JavaScript

```
(true && true && true)
```

---

or

**

Synonyms: `or?`, `||`

```
(or arg1 arg2...)
```

Takes 2 or more arguments and returns true if any of them evaluate to true.

Hot Cocoa Lisp

```
(or false false true)
```

JavaScript

```
(false || false || true)
```

---

xor

**

```
 (xor arg1 arg2)
```

Takes 2 arguments returns true if exactly one of them evaluates to true.

Hot Cocoa Lisp

```
(xor true false)
```

JavaScript

```
((true || false) && (! (true && false)))
```

---

def

Synonyms: `var`

```
(def name [value])
```

Takes 1-2 arguments. Initialize a new variable in the current scope with the specified name. Assign the value to it if one is specified.

Hot Cocoa Lisp

```
(def foo 1)
```

JavaScript

```
var foo = 1;
```

---

set

```
 (set variable [key] value)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value r_value)
```

Assigns the r_value to the l_value and returns the r_value.

If 3 arguments are given:

```
(set object key value)
```

Assigns the specified value to the specified key in the object and returns the value.

Hot Cocoa Lisp

```
(set foo 10)
(set bar baz 12)
```

JavaScript

```
(foo = 10)
(bar[baz] = 12)
```

---

set+

```
 (set+ variable [key] summand)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value summand)
```

Adds the summand to the l_value and returns the new value.

If 3 arguments are given:

```
(set object key summand)
```

Adds the summand to the value associated with the specified key in the object and returns the new value.

Hot Cocoa Lisp

```
(set+ foo 10)
(set+ bar baz 12)
```

JavaScript

```
(foo += 10)
(bar[baz] += 12)
```

---

set-

```
 (set- arg1 arg2...)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value subtrahend)
```

Subtracts the subtrahend from the l_value and returns the new value.

If 3 arguments are given:

```
(set object key subtrahend)
```

Subtracts the subtrahend from the value associated with the specified key in the object and returns the new value.

Hot Cocoa Lisp

```
(set- foo 10)
(set- bar baz 12)
```

JavaScript

```
(foo -= 10)
(bar[baz] -= 12)
```

---

set*

```
 (set* arg1 arg2...)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value factor)
```

Multiplies the l_value by the factor and returns the new value.

If 3 arguments are given:

```
(set object key factor)
```

Multiplies the value associated with the specified key in the object by the factor and returns the new value.

Hot Cocoa Lisp

```
(set* foo 10)
(set* bar baz 12)
```

JavaScript

```
(foo *= 10)
(bar[baz] *= 12)
```

---

set/

```
 (set/ arg1 arg2...)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value divisor)
```

Divides the l_value by the divisor and returns the new value.

If 3 arguments are given:

```
(set object key divisor)
```

Divides the value associated with the specified key in the object by the divisor and returns the new value.

Hot Cocoa Lisp

```
(set/ foo 10)
(set/ bar baz 12)
```

JavaScript

```
(foo /= 10)
(bar[baz] /= 12)
```

---

set%

```
 (set% arg1 arg2...)
```

Takes 2-3 arguments.

If 2 arguments are given:

```
(set l_value modulus)
```

Does modular arithmetic to the l_value and returns the new value.

If 3 arguments are given:

```
(set object key modulus)
```

Does modulus arithmetic to the value associated with the specified key in the object and returns the new value.

Hot Cocoa Lisp

```
(set% foo 10)
(set% bar baz 12)
```

JavaScript

```
(foo %= 10)
(bar[baz] %= 12)
```

++

Synonyms: `inc`

```
(++ number)
```

Takes 1 argument and increments it.

Hot Cocoa Lisp

```
(++ number)
```

JavaScript

```
number++
```

–

Synonyms: `dec`

```
(-- number)
```

Takes 1 argument and decrements it.

Hot Cocoa Lisp

```
(-- number)
```

JavaScript

```
number--
```

---

let

*

```
(let (assignments...) statements...)
```

Takes 2 or more arguments. There must be an even number of assignments. The second of each pair of assignments is evaluated and assigned to the first in a new scope. Then the statements are evaluated in that scope. The result of the last statement is returned.

Hot Cocoa Lisp

```
(let (a 1 b 2)
    (+ a b))
```

JavaScript

```
(function(a, b) { return (a + b); }).call(this, 1, 2)
```

---

* Synonyms: `lambda`, `function`

```
(# (args...) statements...)
```

Takes 2 or more arguments. Creates a new function literal with the specified arguments and statements.

If the final argument specified has three dots after it then it is a splat and all of the remaining arguments passed are assigned to it in an array.

Hot Cocoa Lisp

```
(# (a b) (* a b))

(# (args...) (map args +1))
```

JavaScript

```
(function(a, b) {  return (a * b); })

(function() {var args = [].slice.call(arguments, 0);  return map(args, function(x){return x
```

---

nil?

**

```
 (nil? arg)
```

Takes 1 argument and returns true if it is `null` or `undefined`.

Hot Cocoa Lisp

```
(nil? foo)
```

JavaScript

```
(foo === null || foo === undefined)
```

---

boolean?

**

```
 (boolean? arg)
```

Takes 1 argument and returns true if it is `true` or `false`.

Hot Cocoa Lisp

```
(boolean? foo)
```

JavaScript

```
(typeof(foo) === "boolean")
```

---

number?

**

```
 (number? arg)
```

Takes 1 argument and returns true if it is a number.

Hot Cocoa Lisp

```
(number? foo)
```

JavaScript

```
(typeof(foo) === "number" && (! isNaN(foo)))
```

---

string?

**

```
 (string? arg)
```

Takes 1 argument and returns true if it is a string.

Hot Cocoa Lisp

```
(string? foo)
```

JavaScript

```
(typeof(foo) === "string")
```

---

list?

**

Synonyms: `array?`

```
(list? arg)
```

Takes 1 argument and returns true if it is an array.

Hot Cocoa Lisp

```
(list? foo)
```

JavaScript

```
(Object.prototype.toString.call(foo) === "[object Array]")
```

---

object?

**

```
 (object? arg)
```

Takes 1 argument and returns true if it is an object.

Hot Cocoa Lisp

```
(object? foo)
```

JavaScript

```
(Object.prototype.toString.call(foo) === "[object Object]")
```

---

re?

**

Synonyms: `regex?`, `regexp?`

```
(re? arg)
```

Takes 1 argument and returns true if it is a regular expression.

Hot Cocoa Lisp

```
(re? foo)
```

JavaScript

```
(Object.prototype.toString.call(foo) === "[object Regexp]")
```

---

function?

**

Synonyms: `lambda?`, `#?`

```
(function? arg)
```

Takes 1 argument and returns true if it is a function.

Hot Cocoa Lisp

```
(function? foo)
```

JavaScript

```
(typeof(foo) === "function")
```

---

empty?

**

```
 (empty? arg)
```

Takes 1 argument and returns true if it is `null` or has a length of 0.

Hot Cocoa Lisp

```
(empty? foo)
```

JavaScript

```
(foo === null || (foo).length === 0)
```

---

integer?

**

```
 (integer? arg)
```

Takes 1 argument and returns true if it is an integer.

Hot Cocoa Lisp

```
(integer? foo)
```

JavaScript

```
(typeof(foo) === "number" && foo % 1 === 0)
```

---

even?

**

```
 (even? arg)
```

Takes 1 argument and returns true if it is divisible by 2.

Hot Cocoa Lisp

```
(even? foo)
```

JavaScript

```
(foo % 2 === 0)
```

---

odd?

**

```
 (odd? arg)
```

Takes 1 argument and returns true if it is an integer and not divisible by 2.

Hot Cocoa Lisp

```
(odd? foo)
```

JavaScript

```
(foo % 2 === 1)
```

---

contains?

**

```
 (contains? array value)
```

Takes 2 arguments. Returns true if the array contains the value.

Hot Cocoa Lisp

```
(contains? [ 1 2 3 ] 2)
```

JavaScript

```
([1, 2, 3].indexOf(2) !== -1)
```

---

type

**

Synonyms: `typeof`

```
(type arg)
```

Takes 1 argument and returns a string specifying its type. The types are null, undefined, nan, boolean, number, string, array, object, regex, and function.

Hot Cocoa Lisp

```
(type foo)
```

JavaScript

```
(function(_value_, _signature_) { return ((_signature_ === "[object Array]") ? "array" : (_s
```

---

string

**

```
 (string arg)
```

Takes 1 argument and converts it to a string.

Hot Cocoa Lisp

```
(string 10) ; "10"
```

JavaScript

```
(10).toString() // "10"
```

---

number

**

```
 (number arg)
```

Takes 1 argument and converts it to a number.

Hot Cocoa Lisp

```
(number "10") ; 10
```

JavaScript

```
parseFloat("10") // 10
```

---

integer

**

```
 (integer arg1)
```

Takes 1 argument and converts it to an integer.

Hot Cocoa Lisp

```
(integer "10.1") ; 10
```

JavaScript

```
Math.floor(parseFloat("10.1"))
```

---

re

Synonyms: `regex`, `regexp`

```
(re expression [options])
```

Takes 1-2 arguments. Creates a regular expression literal from the specified string and options. Only one backslash is needed to escape special characters in the expression.

Hot Cocoa Lisp

```
(re "foo" "m")
(re "foo\b")
```

JavaScript

```
(new RegExp("foo", "m"))
(new RegExp("foo\\b"))
```

---

replace

**

```
 (replace subject search replace)
```

Takes 3 strings. Returns a copy of the first with all instances of the second replaced with the third.

Hot Cocoa Lisp

```
(replace "1.800.555.5555" "." "-") ; "1-800.555.5555"
(replace "1.800.555.5555" (re "\." "g") "-") ; "1-800-555-5555"
```

JavaScript

```
"1.800.555.5555".replace(".", "-") // "1-800.555.5555"
"1.800.555.5555".replace((new RegExp(".", "g")), "-") // "1-800-555-5555"
```

---

format

**

```
 (format format-string replacements)
```

Takes a string and either a list or an object. Replaces each instance of `~~` in the format string with the value in the replacements list corresponding to the index of that `~~` (for example the 0th `~~` will be replaced by the 0th element of the replacements list). Replaces each instance `~foo~` in the format string with the value in the replacements object associated with the key `"foo"` (where foo could be any string that contains no tildes.

Hot Cocoa Lisp

```
(format "(~~) (~~) (~~)" [ 1 7 19 ] ) ; "(1) (7) (19)"
(format " *~stars~* _~underbars~_ " { stars "foo"
                                      underbars "bar" } ) ; " *foo* _bar_ "
```

JavaScript

```
var format = function(f,v){var i=0;return f.replace(/~([a-zA-Z0-9_]*)~/g,function(_,k){if(k=
format("(~~) (~~) (~~)", [1, 7, 19]); // "(1) (7) (19)"
format(" *~stars~* _~underbars~_ ", { "stars": "foo", "underbars": "bar" }); // " *foo* _ba
```

---

size

**

Synonyms: `length`, `count`

```
(size arg)
```

Takes 1 argument and returns its size.

Hot Cocoa Lisp

```
(size [ 1 2 3 ] ) ; 3
```

JavaScript

```
[1, 2, 3].length // 3
```

---

compile

```
 (compile file)
```

Takes a path to a file and runs the hcl compiler on that file. Returns the path to the compiled JavaScript file. This is useful for dependency management. It is worth noting that the Node.js module system allows for cyclic dependencies. The behavior of `require` in the cyclic case is document at (http://nodejs.org/api/modules.html#modules_cycles)[http://nodejs.org/api/modules.html#modules_cycles]. `compile` makes sure to only recompile a given file at most once per compilation to allow for these cyclic dependencies.

Hot Cocoa Lisp

```
(require (compile "foo.hcl"))
```

JavaScript

```
require("foo.js") // guaranteed up to date
```

---

from-js

*

```
(from-js identifier)
```

Takes an identifier and inserts it into the compiled source verbatim. Throws an error if the identifier isn't a valid Javascript identifier.

Hot Cocoa Lisp

```
(from-js foo_bar)
```

JavaScript

```
foo_bar
```

References

1. http://www.crockford.com/javascript/javascript.html
2. http://json.org/