JSLisp

Version 0.0.1

Documentation

Table of Contents

Introduction

JSLisp is an implementation of a Lisp dialect as a compiler that generates Javascript code. It has both strong similarities and strong differences with Common Lisp, after which is anyway modeled for the main structure.

The main rule I followed was however to respect the underlying Javascript runtime, so when Javascript and Common Lisp did not agree on a certain basic feature (e.g. string mutability) the choice has been to follow Javascript.

In my opinion Common Lisp is a nice language, but many of the choices made are there just because of historical accidents, and not because of a well thought design process. Common Lisp is a standard, the result of the work of a Committee and this means that it shouldn't surprise that pure logic is not the only force behind a few decisions.

I'm not a Common Lisp expert however, so may be a few divergences in this implementation are indeed just mistakes, time will judge.

JSLisp is for now just an experiment to marry the flexibility of Lisp idea with the WEB world and the efficiency of modern runtime environments for Javascript. Efficiency is what surprised me most... when writing small toy examples the execution speed is closer to highly optimize compilers like SBCL than to interpreters like CLisp.

License

JSLisp is covered by a MIT license:

Similarities with Common Lisp

JSLisp is a LISP-2 (actually a LISP-3 as described in detail later). A symbol in the "*function position*" of a form will be looked up differently than a symbol in a "*value position*" of a form.

Therefore JSLisp in this is closer to Common Lisp than to Scheme.

JSLisp implement unhygienic macros with `defmacro`, with the meaning that it is possible in a macro to capture or introduce new bindings from the expansion context.

This is a powerful feature sometimes and writing practically hygienic macros is not very hard. The code in a macro is just full Lisp, with no compromises... the idea of template-based only macro expansion is in my opinion substantially broken and against the Lisp freedom model.

JSLisp allows to customize the *read* process (even if a different approach from Common Lisp).

JSLisp implements `&rest` and `&key` argument support for functions and macros.

JSLisp scoping rules are those of Common Lisp (e.g. for `let`, `let*`, `do...`, `labels`, `macrolet`, `symbol-macrolet`) and not the ones from Javascript.
This often require wrapping a portion of code in a function construct during compilation, but this is common pattern used also when writing directly in Javascript because native scoping rules are just horrible.

JSLisp symbols can contain any character. This is implemented by mangling and unmangling as needed. The global Javascript namespace is used for symbol's *value*, *function*, and *macro* cells but with names that contains a double dollar sequence to avoid clashing.

JSLisp supports both *lexical* (default) and *dynamic* scoping. Dynamic scoping is only for variables that have been defined with `defvar`.

JSLisp relational operators are all variadic. The "different" `(/= ...)` operator uses the same Common Lisp semantic (the meaning is "*no two operands are equal*", logically different from the negation of the "equal" `(= ...)` operator when more than two operands are used).

Differences from Common Lisp

JSLisp is also very different from Common Lisp, mainly for two reasons:

1. When the underlying Javascript runtime environment supported a different view on a specific topic and when implementing Lisp view (of course always possible) would mean a *serious* impact on the efficiency.
2. When in my eyes the choice of Common Lisp was just an arbitrary historical incident and not a structural logical choice.
   Common Lisp comes from the unification of many Lisp dialects and in many cases the choices made are because of implementation details of those dialects.
   Given that JSLisp (because of 1) doesn't want to be a standard Common Lisp implementation there is no need to feel too much constrained by those choices.
   I'm of course always well open to explanations or suggestion about why it would be more logical to put JSLisp back on Common Lisp track on specific issues.
   Just note however that “*because Common Lisp does so*” for me is not a very good reason per se (in all honesty not a negative, but not a strong positive).

The main differences are:

1. Lists are represented using Javascript arrays. This means there are no `CAR`/`CDR` functions and no distinct vector type. Yes. I'm serious. It's a Lisp *without* cons cells.
   There is a rest function that compiles to `.slice(1)` and that therefore returns a *copy* of the “rest” of a list *without sharing on the top level*.
   Another implication is that `push` is in JSLisp justs a regular function and doesn't need to be a macro (I lied, push is also a macro for efficiency reasons, see later).
2. The numeric and operator support is from Javascript runtime. This means that all numbers are basically double-float of Common Lisp and that therefore there is no support for fixnum, fractions, abitrary precision integers or complex. Addition (+) operator will add numbers and concatenate strings.
3. Native object types are:
   - Numbers (Javascript numbers, what Common Lisp calls `double-float`).
   - Strings (immutable, with unicode support)
   - Lists (Javascript arrays)
   - JS Objects (usable as Common Lisp hash-tables, but keys are strings)
   - `true`, `false`, `NaN`, `null` and `undefined` (with all their strange rules from Javascript). *There is no character type*, even if the reader knows the `#\X` syntax with the meaning of a single-char string (not sure if I'll keep this redundancy).
4. There is no multiple value support
5. There is (so far) no macro destructuring parameters syntax

6. In JSLisp a macro and a function *can have the same name*. This is technically possible because when doing the lookup in the *function position* of a form first the *macro namespace* is checked and, failing that, the *function namespace* is checked after.
   This is similar to what is possible with `define-compiler-macro` of Common Lisp (a feature that I've been told it's seldom used at application level programming).
7. There is a special `defmacro/f` macro that defines at the same time a *macro* and a *function* with the same name (it works only if the function accepts a fixed number of arguments and to get the functions simply calls the macro expansion code with each parameter bound to the same symbol).
8. JSLisp implements `setf` specializations but in a much simpler and naive way than Common Lisp. Many be I will implement the place expander logic in a future, for now when processing `(setf (xxx ...) yyy)` JSLisp simply checks to see if there's a `(set-xxx ...)` function or macro defined to do the assignment.
   Failing that at a last resort it's checked if `xxx` is a macro and in that case it's expanded.
9. String literals are multiline as in Common Lisp and require double-quotes, but they support the escape sequences of Javascript (e.g. `\n`, `\x04`, `\u36BF`).
10. In JSLisp relational operators (`<`, `<=`, `=`, `>=`, `>`, `/=`) are all *both macros and functions* and the macro expansion is *short-circuiting*.
    This means that `(< (foo) (bar) (baz))` will not call `(baz)` if the result is known to be false from the first comparison. This is what the *Python* language does and I also think it's a logical thing to do.
11. There are no declaration forms (for now). May be they will be implemented in a future because they could be used to add checking/debugging code to the generated Javascript. There's however probably nothing to be gained in terms of execution speed from type declarations because the Javascript runtime is inherently dynamically typed.

Javascript integration

JSLisp is a compiler, not an interpreter and every function or lambda defined in JSLisp is a regular Javascript function.

To call a JSLisp global function or to set a JSLisp symbol value or function just use the mangled name. For example to set the JSLisp symbol value of `lisp-var-1` to the number 42 you need to execute the Javascript statement:

```
d$$lisp_var_$49$ = 42;
```

The name mangling rules are:

1. add two dollar signs at the beginning
2. replace any character not in `[-A-Za-z]` with the character code in decimal wrapped between two dollar signs `$`.
3. replace any dash `-` character with an underscore `_` .
4. if you're referring to the symbol value prepend a lowercase `d` to the result, if you're referring to the symbol function prepend a lowercase `f` and if referring to the symbol macro use a lowercase `m`.

To embed Javascript code in JSLisp you can simply use the predefined `(js-code ...)` operator that requires a *string literal* (not an expression!) and that expands *compile time* to that literal. For example this is how the JSLisp function `last-index` could be defined:

```
(defun last-index (x L)
  (js-code "d$$L.lastIndexOf(d$$x)"))
```

this is the Javascript code that this code will be compiled to

```
f$$set_symbol_function(s$$last_index,
  function(d$$x,d$$L){
    var res=d$$L.lastIndexOf(d$$x);
    return res;
  });
```

When executed the above code will set the global `f$$last_index` to the function and therefore from Javascript it will be possible to just evaluate `f$$last_index(a, b)`.

Note that the same mangling `d$$x` is used for both *dynamic* and *lexical* bindings.

It's also possible to call functions with *keyword parameters* by using for example `s$$$58$x` to reference the symbol `:x` in the call from Javascript.

Reference

Data types

Numbers

These are Javascript native number objects (similar to Common Lisp `DOUBLE-FLOAT` type). There are no fixnum or arbitrary precision integer numbers but the type unit accuracy is guaranteed up to $\pm 2^{53}$ (±9,007,199,254,740,992).

There is a special numeric value `NaN` that is generated when a computation is not possible (e.g. evaluating `(log -1)`). `NaN` value compare different from any number (including `NaN`) and any other relational operation with `NaN` returns false (e.g. are both false `(< x NaN)` and `(>= x NaN)` for any value of `x`).

The predicate function `(numberp x)` returns `true` if `x` is a number (including `NaN`) and `false` otherwise.

Strings

Immutable Javascript unicode strings. In string literals JSLisp supports standard Javascript escape sequences for *octal*, *hexadecimal* or *unicode characters* in addition to standard special character escape sequences (tab, newline, carriage return, backspace, formfeed).

There is no *character type*, characters of strings are accessed using `(aref <string> <index>)` form (with a *0-based* index) and the value is a *single-character string*.
The predicate function `(stringp x)` returns `true` if `x` is a string and `false` otherwise.

Symbols

There is (currently) *no package support*, but when a symbol starting with a *colon* `:` character is interned its value is also *set to the symbol itself*.

These symbols can be used for keyword parameters in function calls and mimic the keyword package of Common Lisp.

The predicate function `(symbolp x)` returns `true` if `x` is a symbol and `false` otherwise.

Lists

Lists are Javascript arrays and element access is with `(aref <list> <index>)` using a *0-based index*. List tails are *never shared*, the predefined `(rest <list>)` function returns therefore a *copy* of the list starting from second element.

JSLisp lists are more or less equivalent to Common Lisp `vector`s, and for example `(push <element> <list>)` can be just a regular function and doesn't need to work on a *place* like it's required in Common Lisp.

When accessing a list element that doesn't exist (because beyond the size of the list) the returned value is `undefined`. When *writing* to that element the list object is *automatically resized* and any intermediate values are all set to `undefined`.

The predicate function `(listp x)` returns `true` if `x` is a list and `false` otherwise.

Closures

These are just regular Javascript callable objects. They're used as the result of evaluating `(lambda ...)` forms.

Being JSLisp is a *compile-only implementation*, there's no difference between the result of a lambda form and a manually written Javascript function.

The `(eval ...)` function is also available but is implemented as *compile+invoke* and not as an *interpreter*. Because `eval` is a regular function and not a *macro* the evaluation will not be able to access *lexical bindings*.

Boolean

Javascript boolean type with only values being `true` and `false`.

Following Javascript rules an expression is considered "true" unless its value is either:

- `false`, `null`, `NaN`, `undefined`
- The number `0`
- The empty string `""`

Note that *empty lists* or *empty Javascript objects* are considered to be “*true*” (this is for example different from what the Python language does).

The function `(boolp x)` returns `true` if a value is a boolean and `false` otherwise.

Null

Javascript null type with only one value: `null`. This value is used as result for some operations like division by zero. Note that `(= null undefined)` is true (I've no idea why).

The function `(nullp x)` returns `true` if a value is the *null value* or `false` otherwise.

Undefined

Javascript undefined type with only one value: `undefined`. This value is used when there is no reasonable value to return. For example accessing an array or string outside its size will return `undefined` on reading. Likewise reading a Javascript object field that doesn't exists returns `undefined`.

The function `(undefinedp x)` returns `true` if `x` is the `undefined` value and `false` otherwise.

Objects

Javascript native objects. They can also used as hash tables where keys are only strings (on `aref` access the key is automatically converted to a string using Javascript conventions).

For example:

```
(let ((x (js-object)))
  (setf (aref x "12") "Test")
  (aref x 12))

--> "Test"
```

The function `(objectp x)` returns `true` if `x` is a Javascript object and `false` otherwise.

Namespaces

There are three main distinct namespaces; the value namespace, the function namespace and the macro namespace.

Value namespace

This namespace contains the current value associated to a global symbol. This value can be accessed or altered by using the functions:

```
(symbol-value x)        ;; Retrieves current value of symbol x
(set-symbol-value x y) ;; Sets current value for symbol x
```

In the Javascript runtime the value of a symbol `x` is stored in the global variable named "`d<mangled-name>`". For example for symbol `x1` the variable is `d$$x$49$`.

Function namespace

This namespace contains the current function associated to a global symbol. It can be accessed or altered by the two functions

```
(symbol-function x)        ;; Current function of symbol x
(set-symbol-function x y) ;; Sets function for symbol x
```

In the Javascript runtime the function of a symbol `x` is stored in the global variable named "`f<mangled-name>`". For example for symbol `foo` the variable is `f$$foo`.

Macro namespace

This namespace contains the current macro associated to a global symbol. It can be accessed or altered by the two functions

```
(symbol-macro x)       ;; Retrieves current macro of symbol x
(set-symbol-macro x y) ;; Sets macro expander for symbol x
```

In the Javascript runtime the macro of a symbol `x` is stored in the global variable named "`m<mangled-name>`". For example for symbol `<` the variable is `m$$$60$`.

The macro value of a symbol is used by the compiling function (js-compile ...) when a symbol is found in the function position of a form. In this case the logic is:

1. If there is a lexical macro currently visible with that name then it is expanded
2. If there is a global macro with that name then it is expanded
3. Otherwise the code for calling a function associated to the symbol is generated. For example the Lisp code `(foo x)` in this case will generate the Javascript code `(f$$foo(d$$x))`. Note also that `f$$foo` could actually be bound to a *lexical function* because of a containing `(labels ...)` form.

Macro expansion is done at compilation time. After a form is compiled defining a macro for the first symbol in the form will have no effect on already compiled functions; redefining instead the function associate to a symbol will affect also previously compiled function where the call has been compiled to a global function call.

When a `(labels ...)` form is compiled any *global* or *lexical* macros bound to function names defined in the labels part will not be considered. In other words...

```
(defmacro foo (x y)
  `(+ ,x ,y))

(let ((x 3)
      (y 4))
  (list (labels ((foo (x y) (* x y)))
          (foo x y))
        (foo x y)))

--> (12 7)
```

Flow control

(progn ...)

```
(progn
  <form-1>
  <form-2>
  …
  <form-n>)
```

The `progn` form evaluates all contained forms in sequence and returns as value the value of the last evaluated form.

(dotimes …) (dolist ...)

```
(dotimes (<var> <count>)
   <form-1>
   <form-2>
   …
   <form-n>)
```

The `dotimes` form evaluates all the contained forms in sequence by first assigning the value `0` to the variable `<var>`, then `1`, then `2` and so on until the variable would get the value of `<count>`. At this point (without evaluating the forms with `<var>` equal to `<count>`) the loop is terminated and the final value is `null`. The form `<count>` is evaluated only once and the body of the loop *may be not executed* if the result of evaluating `<count>` is less than `1`.

```
(dolist (<var> <list>)
  <form-1>
  <form-2>
  …
  <form-n>)
```

The `dolist` form evaluates all the contained forms in sequence by first assigning to the variable `<var>` the first element of the list returned by evaluating `<list>`, then the second and so on until the end of the list.
Once the list is exhausted the loop is terminated and the final value is `null`. The form `<list>` is evaluated only once and the body of the loop *may be not executed* if the result of evaluating `<list>` is an empty list.

(do ...)

```
(do ((<var-1> <init-1> [<next-1>])
     (<var-2> <init-2> [<next-2>])
     …
     (<var-n> <init-n> [<next-n>]))
    (<end-test>
     <end-form-1>
     <end-form-2>
     …
     <end-form-n>)
  <body-form-1>
  <body-form-2>
  …
  <body-form-n>)
```

The `do` form is the most general looping facility provided by JSLisp and is modeled after the standard do of Common Lisp.

Several distinct looping variables `var-1`, `var-2`, ... `var-n` can be defined for the loop in the first sublist, for each of them the *initial value* must be provided and optionally a *next value expression*. The *next value expression* for a variable can be omitted if the variable doesn't need to change during the loop or if the increment is done *explicitly* in the *body part*.

After the looping variables is specified the `end-test` form (that must evaluate to true to exit the loop) and the closing sequence of `end-form`s. The final value of a `do` construct will be the value of the last `end-form` or `null` if no end forms are present.

Finally there is a list of `body-form`s that will be evaluated for each iteration after checking that the `end-test` is not passing. The evaluation of the `next-...` forms will be done after the evaluation of the last `body-form`.

As an example consider this simple function for checking for primality of an integer >= 2 (note that here the loop body is actually empty - a fairly common case):

```
(defun prime-number (n)
  (cond
     ((= n 2) true)
     ((= 0 (% n 2)) false)
     (true
        (do ((x 3 (+ x 2)))
            ((or (> (* x x) n)
                 (= 0 (% n x)))
             (> (* x x) n))))))
```

(if …) (when …) (unless ...)

```
(if <condition>
    <if-true-form>
    [<if-false-form>])
```

The `if` form allows conditional execution. After evaluating the *condition form* only one of the two subsequent forms will be evaluated depending on if the condition passed or not.

If the `if-false-form` is omitted the final value when the condition is not met is `null`.

```
(when <condition>
  <body-form-1>
  <body-form-2>
  …
  <body-form-n>)
```

```
(unless <condition>
  <body-form-1>
  <body-form-2>
  …
  <body-form-n>)
```

These two forms are *specialized versions* of the `if` form. In many cases what is needed are multiple operations in one case and nothing in the other. To simplify writing and reading of Lisp code the two forms are equivalent to `if`+`progn` constructs for these cases.

The `when` form evaluates the `body-form`s if the condition is passed, the `unless` form instead evaluates them if the condition is *not* passed.

In both the final value is either `null` or the value of `body-form-n`.

(let …) (let* ...)

```
(let ((<var-1> <value-1>)
      (<var-2> <value-2>)
      ...
      (<var-n> <value-n>))
  <form-1>
  <form-2>
  ...
  <form-n>)

(let* ((<var-1> <value-1>)
       (<var-2> <value-2>)
       ...
       (<var-n> <value-n>))
  <form-1>
  <form-2>
  ...
  <form-n>)
```

Both `(let ...)` and `(let* ...)` forms can define lexically scoped variables for the use inside the body. In case of `(let ...)` all the initialization forms are evaluated in the enclosing lexical environment and only when all the computation have been completed the new symbols are bound to the values. In `(let* ...)` instead every variable definition can see all preceding symbols as already bound to their value, allowing for cascading initializations.

In both cases once all the variables have been initialized the *body forms* `form-1`, `form-2`, ... `form-n` are evaluated in sequence and the value of the last form becomes the value of the whole `(let ...)` form.

A common idiom is `(let ((x x)) ...)`. This can be necessary in case of loops if the variable being looped on is *just assigned* and not *rebound* at each iteration. For example:

```
(let ((L (list)))
  (dotimes (i 10)
    (push (lambda (x) (* x i)) L))
  (map (lambda (f) (funcall f 7)) L))
--> (70 70 70 70 70 70 70 70 70 70)
```

In this case all the *closures* saved in the list `L` are indeed bound *to the same lexical variable* `i` that simply changed its *value* at each iteration.

```
(let ((L (list)))
  (dotimes (i 10)
    (let ((i i))
      (push (lambda (x) (* x i)) L)))
  (map (lambda (f) (funcall f 7)) L))
--> (0 7 14 21 28 35 42 49 56 63)
```

In this case instead the `(let ((i i)) ...)` form actually creates a new lexical variable `i` bound to the current value of the *external* variable `i` used in the loop and each closure will have *its own separate variable*.

To see the difference between `(let )` and `(let* ...)` consider the following examples:

```
(let ((x 10))
  (let ((x 20)
        (y (1+ x)))
    (list x y)))
--> (20 11)
```

In the `(let ...)` case the initialization of y uses the *external* `x` binding, so we get as value for that variable `11`.

```
(let ((x 10))
  (let* ((x 20)
         (y (1+ x)))
    (list x y)))
--> (20 21)
```

In the `(let* ...)` case instead the initialization of `y` can already "see" the previous binding and therefore the value will be `21`.

Note that differently from C/C++ even in this second case the form used to initialize the variable `x` will not see the `x` variable itself already. So for example with:

```
(let ((x 10))
  (let* ((x (lambda () (* x 2)))
    ... ))
```

the closure will *capture* the external `x` variable, even if `(let* ...)` has been used.

You can still create a variable holding a closure that captures the variable itself (sort of a self-referential closure) creating first the variable, then the closure referring to it and the assigning the closure to the variable.

Predefined functions and macros

(defun ...) (labels ...)

(defmacro ...) (defmacro/f ...) (macrolet ...) (symbol-macrolet...)

(< ...) (<= ...) (>= ...) (> ...) (= ...) (/= ...) (== ...)

(and ...) (or ...)

(+ ...) (- ...) (* ...) (/ ...) (% ...)

(ash x count) (logcount x) (logand ...) (logior ...) (logxor ...)

(boolp x) (undefinedp x) (nullp x) (NaNp x) (objectp x) (zerop x) (numberp x) (stringp x) (symbolp x) (listp x)

(length seq) (aref seq index/key) (set-aref seq index/key value)

(list ...)

(first x) (second x) ... (tenth x)

(rest x) (slice x from to)

(reverse x) (nreverse x)

(sort seq pred) (nsort seq pred)

(setf place value) (incf place delta) (decf place delta)

(1+ x) (1- x)

(min seq) (max seq)

(reduce op seq)

(map f seq)

(zip ...)

(mapn f ...)

(make-array)

(filter f seq)

(range x y z)

(index x seq)

(last-index x seq)

(defstruct ...)

(sin x) (cos x) (tan x) (exp x) (log x) (atan x) (floor x) (abs x) (atan2 y x) pi

(try ...)

(clock) (time ...)

(. obj ...) (js-object)

Libraries

Ajax support

(ajax url callback error-callback)

DOM interface

document window (get-element-by-id) (create-element)

(append-child parent child) (remove-child parent child)

(set-style element ...)

(tracking x y f)

(dragging element x y)

GUI

(window x y w h title &key close resize)

(button text action)

Automatic layout (:V ...) (:H ...) (:Vdiv ...) (:Hdiv ...) (set-coords node x0 y0 x1 y1)

Reader customization

(reader-function x) (set-reader-function x) (parse-value str)

Compiler interface

(js-code literal) (js-compile form) (js-eval string)

(compile-specialization symbol) (set-compile-specialization symbol code)

The REPL

Browser compatibility

Basic editing

Permanent sessions

Examples

Implementation

Implementation of namespaces

JS-Compile

Bootstrap

Test

Compatibility issues