Superscript Language Proposal

Uday Singh, Samurdha Jayasinghe, Tommy Orok, Yu Wang, Michelle Zheng
(urs2102, sj2564, to2240,yw2684, myz2103)

Motivation

"JS is the x86 of the web" - Brandon Eich

Why compile to Javascript? As Brandon Eich said, "JS is the x86 of the web"[0] . Today Javascript is used not only to govern interactions on the front end of the internet, but also to prototype systems on the backend of the web.

In recent years, the web has transformed in terms of interaction design and web browser functionality. The modern web has progressed from static pages and now demands a system[1] that provides reactions based on data changes. This necessity for real-time interaction has led to a shift in the world of rapid prototyping, from monolithic web frameworks, to a clear separation between single-page applications and APIs.

The development of Node.js has led to more developers embracing Javascript as a viable language for backend development. Despite its popularity however, Javascript's verbose syntax, mutable objects, illogical equality comparisons, and complex callbacks make it difficult for developers be completely aware of their code's side effects[2].

Developers should understand the full semantic structure of their code, especially when prototyping web servers and RESTful APIs, and dealing with callbacks and Node.js requests. However, closures and anonymous functions in Javascript are often confusing to write and understand. The number of lines of code is also directly correlated with the number of bugs generated.

Despite Javascript treating functions as first class objects, Javascript is typically used as an imperative programming language. Its lack of structure has encouraged programmers to think of Javascript objects in terms of state, instead of attempting to transform data. Javascript is a product of rapid evolution, and thus for many people from the functional programming school of thought, it seems broken. Although its core library is small, Javascript's weak typing and object construction system create a broken mix, where functions are first class, yet objects are used imperatively.

Our solution is Superscript, a type-inferred language, inspired by a mix of Lisp, Clojure and Arc that compiles to Javascript.

Language Description

"Lisp isn't a language, it's a building material." - Alan Kay

Superscript is a Lisp focused on rapid development to compile to Javascript. Primarily being heavily influenced from both Arc and Clojure, two very new languages introduced to the Lisp community. Superscript is a Lisp designed to have very intentional syntax, allowing the user to know very little to write very large programs.

Unlike Javascript, Superscript encourages users to write in a functional first language thinking more before they code. Using an inferred type system, similar to Ocaml, our compiler tells users when their functions break for particular type and strongly encourages functional thinking over imperative thinking. This breaks users from thinking in terms of objects as many new programmers do when looking at Javascript and encourages clean data transformations which the user is aware of through s-expressions. This, combined with the flexibility of the Lisp family of languages, where functions and data are equivalent, gives users of Superscript a level of power unavailable in languages like Javascript.

Object-oriented programming tends to be written with a lack of discipline on the developer's' part, by preventing them from doing much damage by abstracting everything. Superscripts focus on power and brevity encourages users for rapid prototyping to write succinct code, which results in clean, type-inferred functions in Javascript. Additionally, another shortcoming of Object-Oriented programming is that often times due to the lack of power provided by object-oriented languages, users tend to believe they are doing more work than they are actually outputting. Superscript encourages thoughtful programming, translated to Javascript that is performant, functional, and well typed.

Superscript uses syntax based on Lisp and will translate to Javascript. It builds upon the primary Lisp functions[3], and in addition, it allows for user-defined functions, recursion, and type inference with seven data types. Superscript's syntax is based on closure, and is built from multiple nested S-expressions. Similar to OCaml, it is functional. Unlike languages like Scheme, it is type inferred. Expressions are evaluated in prefix order. Superscript supports the typical arithmetic and logical operations.

Superscript intends to provide sharp programmers a language to think and express clear functional ideas in a language which is slowly becoming the backbone of the web.

User-Defined Functions ('subscripts')

`(fn '(argument*) expression*)` defines an anonymous function that takes a list of arguments and an expression that will be evaluated to return the result of the function:

```
#(fn '() "Hello World")
-: <Function>
```

(= function_name (fn ‘(argument*) expression*)) binds the anonymous function passed in as the second argument to the name specified by the first argument (function_name).

```
# (= hello (fn ‘() “Hello world”))
-: user/hello
# (hello)
-: “Hello World”
```

Equivalently, (def function-name ‘(argument*) expression*)) is shorthand for (= function-name (fn ‘(argument*) expression*)).

```
# (def hello ‘() “Hello World”)
-: user/hello
# (hello)
-: “Hello World”
```

Recursion

Superscript will support recursion so that functions may call themselves. Examples of recursive Superscript code are provided under Intentional Syntax & Sample Code.

Type Inference

Superscript will support type inference; the data type of an expression will be automatically deduced at compile-time. The compiler will draw conclusions about the types of variables based on how programmers use those variables, similar to OCaml and Haskell. The data types supported by Superscript are shown in the table below.

Superscript Data Types

Data Type	**Example**
int (atom)	`1, 5, -10`
float (atom)	`1.5, 4.0, -200.7`
boolean (atom)	`true, ()`
list	`(a b c), (1 2 3), (true true false)`
nil / empty list / false (atom)	`()`
String (atom)	`“abc”, “ABC”, “cat”`
subscript (procedure or anonymous function)	`(fn ‘() “Hello world”)`

Operators and Built-in Functions

Operator Name	**Syntax**	**Applicable Data Types**
Basic assignment	`(= a b)`	All

Integer arithmetic
Floating point arithmetic
Modulo
Remainder
	`(+ a b), (- a b), (* a b), (/ a b)`
	`(+. a b), (-. a b), (*. a b), (/. a b)`
	`(mod a b)`
	`(rem a b)`
		int
		float
		int
		int

Equal to
Not equal to
Comparison operators
	`(is “a” “b”), (is 1 2)`
	`(isnt “a” “b”), (isnt 1 2)`
	`(> a b), (>= a b), (<= a b)`
		int, float, String
		int, float, String
		int, float, String

Logical AND
Logical OR
Logical NOT
Bitwise operators
	`(and a b)`
	`(or a b)`
	`(not a)`
	`(& a b), (|a b), (^ a b)`
		boolean
		boolean
		boolean
		int

N-th element of list (0-based) Isomorphic list equality	`(get N a)` `(iso a b)`	list
String concatenation	`( + “hello” “world)`	String

Primary Lisp Functions[3]

Operator Name	**Syntax**

`(quote a)` returns `a`. For shorthand, abbreviate it as `‘a`. You must quote values if you do not want the S-expression to be evaluated, but instead to be returned.	`# (quote a)`
	`:- a`
	`# (‘a)`
	`:- a`

`(atom ‘a)` returns the atom **`true`** if the value of `a` is an atom or the empty list.	`# (atom ‘a)`
	`:- true`
	`# (atom ‘(a b c))`
	`:- ()`
	`# (atom ‘())`
	`:- true`

`(is a b)` returns **`true`** if `a` equals `b`. Returns the empty list, equivalent to boolean false, when `a` is not `b`.	`# (is ‘a ‘a)`
	`:- true`
	`# (is ‘a ‘b)`
	`:- ()`

`(head a)` expects `a` to be a list, and returns the first element of `a`.	`# (head ‘(a b c))` `:- a`

`(tail a)` expects a to be a list, and returns all elements after the first element.	`# (tail ‘(a b c))` `:- (b c)`
`(cons a b)` expects the value of b to be a list, and returns a list containing the element a followed by the elements of list b	`# (cons ‘(a) ‘(b c))` `:- (a b c)`
`(if a b c)` where a is an S-expression which returns a boolean; equivalent to `if` a `then` b `else` c.	`# (if (odd 1) ‘a ‘b)` `:- a`
Additional use of if function	**Syntax**

`(if a b c d e)` where a is an expression and b is a return value considers `c` to fall as a condition for an else-if statement with the result of `c` being true resulting in `d` and with the else of the entire if statement being `e.`	`(if a b c d e)` is equivalent to:
	`(if a`
	`b`
	`(if c`
	`d`
	`e))`

Intentional Syntax & Sample Code

GCD

```
;
; Superscript source code for GCD
;

(def gcd (a b)
  (if (is 0 b)a
      (gcd b (mod a b))))
```

```
/**
 * GCD in resulting Javascript
 */

var gcd = function(a, b) {
  if (b === 0) {
      return a;
  }
  return gcd(b, a % b);
};
```

Insertion Sort[5]

```
;
; Insertion Sort
; Superscript source code
;
(def sort (myList)
  (if (is '() myList) '()
  (insert (head myList) (sort (tail myList)))))

(def insert (item myList)
  (if (is '() myList)
  (list item)
      (if (< item (head myList))
      (cons item myList)
          (cons (head myList) (insert item (tail myList))))))
```

Insertion Sort, translated to Javascript:

```
var __clone = function(obj) {
  return JSON.parse(JSON.stringify(obj));
};
var __head = function(list) {
  return __clone(list[0]);
};

var __tail = function(list) {
  return __clone(list.slice(1));
};

var __list = function(item) {
  return [__clone(item)];
};

var __cons = function(item, list) {
  if (list === null) {
      return __list(item);
  } else {
      var __temp = __clone(list);
      __temp.unshift(__clone(item));
      return __temp;
  }
};
```

```
var insert = function(item, myList) {
  if (myList.length === 0) {
      return __list(item);
  } else {
      if (item < myList[0]) {
          return __cons(item, myList);
      } else {
          return __cons(__head(myList), insert(item, __tail(myList)));
      }
  }
};

var sort = function(myList) {
  if (myList.length === 0) {
      return __clone(myList);
  } else {
      return insert(__head(myList), sort(__tail(myList)));
  }
};
```

Recursive list length function in Superscript:

```
;
; Superscript length function
;


(def length (lst)
  (if (no lst)
      0
      (+ 1 (length (tail lst)))))
```

List length function, translated to Javascript:

```
var __clone = function(obj) {
  return JSON.parse(JSON.stringify(obj));
};

var __tail = function(list) {
  return __clone(list.slice(1));
};

var length = function(lst) {
  if (lst.length === 0) {
      return 0;
  } else {
      return 1 + length(__tail(lst));
  }
};
```

Footnotes & References

[0]: Javascript is Assembly Langauge for the Web http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx

[1]: 7 Principles of Rich Web Applications. Guillermo Rauch. http://rauchg.com/2014/7-principles-of-rich-web-applications

[2]: Wat, a lightning talk by Gary Bernhardt. https://www.destroyallsoftware.com/talks/wat

[3]: The Roots of Lisp. Paul Graham. http://languagelog.ldc.upenn.edu/myl/ldc/llog/jmc.pdf

[4]: Arc. Paul Graham. http://www.paulgraham.com/arc.html

[5]: Insertion Sort Lisp Implementation. Bob Dondero. http://cs.princeton.edu/courses/archive/spr11/cos333/lectures/17paradigms/sort.lisp

[6]: Clojure for the Brave and True. Daniel Higginbotham http://www.braveclojure.com/