



A Strongly Typed Language for the Web

Ayush Jain · Gaurang Sadekar · Bahul Jain · Prakhar Srivastav

Just another day in the Javascript world



Christian Rebischke

@Sh1bumi



Follow

The reason why javascript sucks..

```
> '5' - 3
2 // weak typing + implicit conversions * headaches
> '5' + 3
'53' // Because we all love consistency
> '5' - '4'
1 // string - string * integer. What?
> '5' + + '5'
'55'
> 'foo' + + 'foo'
'fooNaN' // Marvelous.
> '5' + - '2'
'5-2'
> '5' + - - + - - + - - + + - + - + - - - '-2'
'52' // Apparently it's ok

> var x * 3;
> '5' + x - x
50
> '5' - x + x
5 // Because fuck math
```

RETWEETS

1,970













LIKES

1,119



Why JSJS?

How does JSJS compare to JS?

Feature	JSJS	JS
Type Safety		
Type Inference		
Immutable values		
Immutable Collections		
Programming Style	Functional	Imperative
Works with Node		
Works in the browser		



A Sneak Peak - Syntax

```
val sort = /\(xs: list T): list T => {
  if xs == [] then []
  else {
    val smaller = List.filter(/\(x) => x < hd(xs), tl(xs));
    val greater = List.filter(/\(x) => x >= hd(xs), tl(xs));
    List.concat(sort(smaller), hd(xs) :: sort(greater));
  };
};

// let's test it
val sorted = sort([10, 5, 0, 3, 8]);
val sorted_strs = sort(["c", "z", "a", "e", "y"]);

// printing...
List.iter(print, sorted);
List.iter(print, sorted_strs);
```



A Sneak Peak - Syntax

```
val sort = /\(xs: list T): list T => {
  if xs == [] then []
  else {
    val smaller = List.filter(/\(x) => x < hd(xs), tl(xs));
    val greater = List.filter(/\(x) => x >= hd(xs), tl(xs));
    List.concat(sort(smaller), hd(xs) :: sort(greater));
  };
};

// let's test it
val sorted = sort([10, 5, 0, 3, 8]);
val sorted_strs = sort(["c", "z", "a", "e", "y"]);

// printing...
List.iter(print, sorted);
List.iter(print, sorted_strs);
```

Annotations in the code:

- Overloaded operators**: points to `==` and `<` / `>=`
- Type Annotations (optional)**: points to `list T`
- Lambda Expressions**: points to `/\(x) =>`
- Immutable values**: points to `List`
- Rich collections library**: points to `List`
- Comments**: points to `// let's test it`
- List Literals**: points to `[10, 5, 0, 3, 8]` and `["c", "z", "a", "e", "y"]`
- StdLib**: points to `List`
- Cons operator**: points to `::`



Functional Programming

Functions everywhere...

```
val map = /\(fn: (T) -> U, xs: list T): list U => {  
  if empty?(xs) then []  
  else fn(hd(xs)) :: map(fn, tl(xs));  
};
```

```
val sq = /\(x) => x * x;
```

```
print(map(sq, [1,2,3,4,5,6]));
```



Closures

Local value for a function - kept alive after function has returned

```
val sayHelloTo = /\(name) => {  
    val text = "Hello " ^ name;  
    /\() => print(text);  
};
```

```
val sayTo = sayHelloTo("Bob");  
sayTo(); // prints "Hello Bob"
```



Error Handling

Error Reporting

```
1 val x = 1 && 2;
```

Type error: expected value of type 'num', got a value of type 'bool' instead

Exceptions and Exception Handling

```
1 try {  
2   throw "You have failed this program...";  
3 } catch (msg) {  
4   print("Maybe not \o/");  
5 };  
6
```

Maybe not \o/



Immutability

- Immutability makes it easier to reason about code
- The `val` keyword defines a value. No **variables** in JSJS.
- Names cannot be redefined in the current scope

```
val x = 5;
// not allowed
val x = 7;
```

Error: 'x' cannot be redefined in the current scope

```
let locals, globals = env in
if NameMap.mem id locals then raise (AlreadyDefined(id))
else if KeywordsSet.mem id js_keywords_set then raise (CannotRedefineKeyword(id))
else
```



Immutable Collections - Lists & Maps



IMMUTABLE

- Uses Facebook's Immutable.js library to enforce immutability.
- All library functions written in JSJS.



Immutable Collections - Lists

```
val xs = [1, 2, 3, 4, 5];  
  
// using cons operator  
val _ = "foo" :: ["bar", "baz"];  
  
// heterogenous collections not allowed  
val _ = [true, "foo", 10];
```

All functions return new lists (do not modify the list in place)

List Collection Library

```
hd, tl, empty?, filter, map, fold_left, rev, iter, range, concat, insert,  
remove, sort, nth, length
```



Immutable Collections - Map

```
val moneyOwedByFriends = {  
  "ben": 10,  
  "mary": 20,  
  "mark": 43,  
  "alice": 54  
};  
  
// map annotations  
val getTotal = /(m: <string: num>): num => {  
  val values = Map.values(m);  
  List.fold_left(/(x, y) => x + y, 0, values);  
};  
  
getTotal(moneyOwedByFriends);
```

All functions return a new map (do not modify the map in place)

Map Collection Library

get, set, has?, length, values, keys, count, merge, del



JSJS Type System

T(_) is a generic type that helps with type Inference or generic user annotations.

TAny is only used for type inference.

TNum, TBool, TUnit, TString are data types.

TFun is the function type comprising of args (list of primitives) and a return type.

TList and TMap are composite List and Map types

```
(* Types in JSJS are either a primitive type
or a function type. both of these types are
defined in a mutually recursive fashion *)
type primitiveType =
  (* a generic type *)
  | T of string
  (* a general type. used to define
  empty lists or empty maps *)
  | TAny
  | TNum
  | TString
  | TBool
  | TUnit
  (* type of exception associated with a message *)
  | TExn
  | TFun of funcType
  (* list type - a list of primitive types *)
  | TList of primitiveType
  (* map type - a tuple of key type, value type *)
  | TMap of primitiveType * primitiveType

(* a function type takes a list of primitive types
and returns a primitive type *)
and funcType = primitiveType list * primitiveType
```



Type Inference - Our 'Inspiration'



Guillermo Rauch

@rauchg



Following

JSJS is a very cool proposal for type-safe JS.
Digging the syntax a lot.
[cs.columbia.edu/~%20sedwards/c ...](https://cs.columbia.edu/~%20sedwards/c)

Composite Data Types

List

List Type Declaration

- The `list` keyword is used to declare the type of a list
- Type declaration for a list: `a : list T`

List Declaration

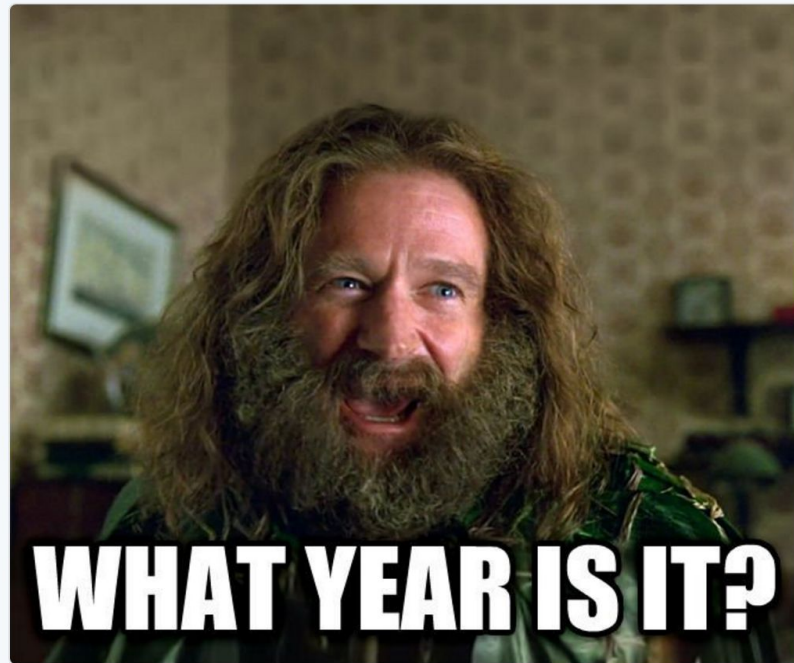
```
// List of strings
val names : list string = ["foo", "bar", "baz"];

// List of nums
val nums : list num = [1, 2, 3, 4];
```



-e^(pi)x engiqueer @FrozenFire · Feb 21

@rauchg tfw someone introduces a new strongly typed functional language without type inference



5



Type Inference - JSJS vs OCaml

JSJS

```
gaurang@dyn-160-39-206-186 ~/C/s/p/JSJS> ./jsjs.out -s  
^(f) => f(42);  
^ (f : (num) -> A) : A = { { (f: (num) -> A)((42.: num)) : A } }
```

OCaml

```
utop # fun f -> f 42;;  
- : (int -> 'a) -> 'a = <fun>
```



Type Inference - JSJS vs OCaml

JSJS

```
^(f => f(f(42)));  
^(f : (num) -> num) : num = { { (f: (num) -> num)((f: (num) -> num)((42.: num)) : num) : num } }
```

OCaml

```
utop # fun f -> f (f 42);;  
- : (int -> int) -> int = <fun>
```



Type Inference with user annotations

```
gaurang@dyn-160-39-206-186 ~/C/s/p/JSJS> ./jsjs.out -s
^(x, y) => x < y;
^ (x : A, y : A) : bool = { { ((x: A) < (y: A): bool) } }
gaurang@dyn-160-39-206-186 ~/C/s/p/JSJS> ./jsjs.out -s
// with annotations
^(x: num, y) => x < y;
^ (x : num, y : num) : bool = { { ((x: num) < (y: num): bool) } }
```



Type Errors - JSJS vs OCaml

JSJS

```
^(x, y) => { print_string(y); x + y;};  
Type error: expected value of type 'string', got a value of type 'num' instead
```

OCaml

```
utop # fun x y -> print_endline y; x + y;;  
Error: This expression has type bytes but an expression was expected of type  
      int
```



Type Errors - JSJS vs OCaml

JSJS

```
^(z) => z(z);
```

Type error: expected value of type 'A', got a value of type '(A) -> B' instead

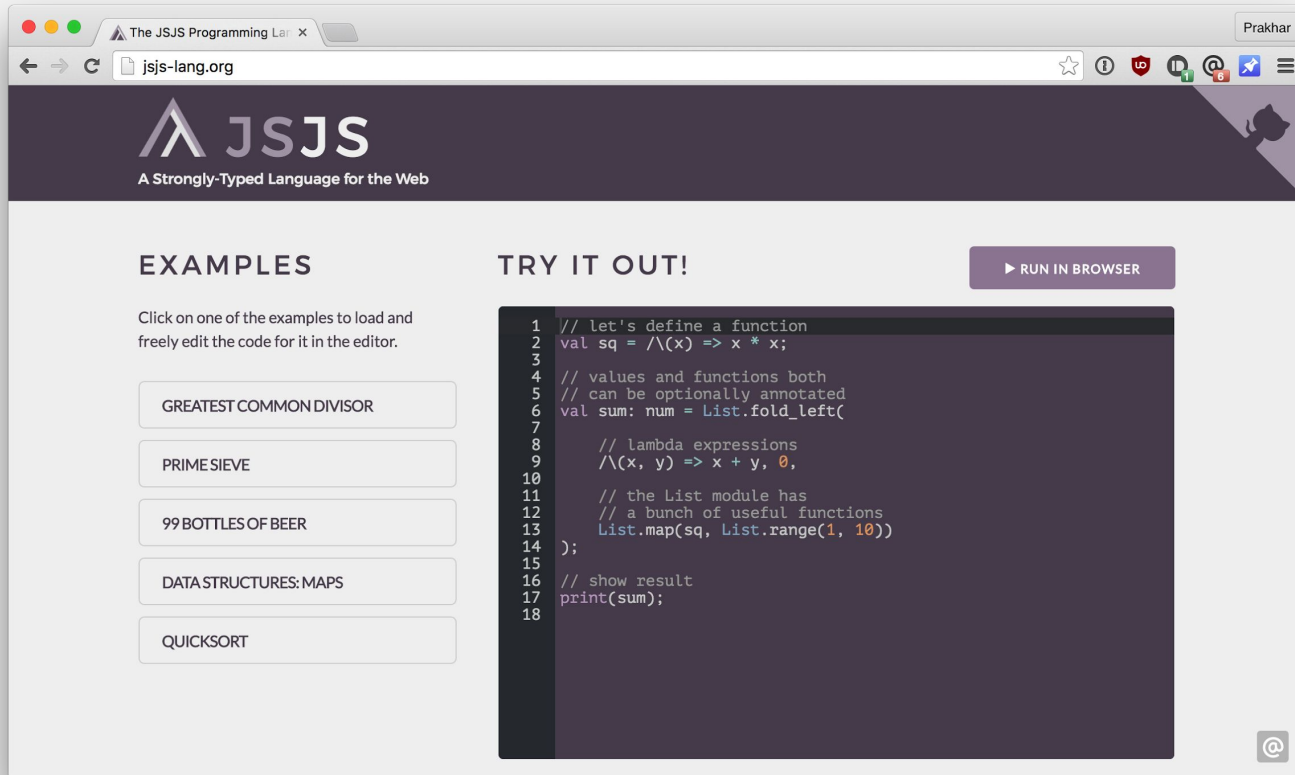
OCaml

```
utop # fun z -> z z;;
```

```
Error: This expression has type 'a -> 'b  
      but an expression was expected of type 'a  
      The type variable 'a occurs inside 'a -> 'b
```



Demo



The screenshot shows a web browser window with the URL `jsjs-lang.org`. The page features the JSJS logo and the tagline "A Strongly-Typed Language for the Web". Under the "EXAMPLES" section, there are five buttons: "GREATEST COMMON DIVISOR", "PRIME SIEVE", "99 BOTTLES OF BEER", "DATA STRUCTURES: MAPS", and "QUICKSORT". To the right, the "TRY IT OUT!" section includes a "RUN IN BROWSER" button and a code editor with the following code:

```
1 // let's define a function
2 val sq =  $\lambda(x) \Rightarrow x * x$ ;
3
4 // values and functions both
5 // can be optionally annotated
6 val sum: num = List.fold_left(
7
8     // lambda expressions
9      $\lambda(x, y) \Rightarrow x + y, 0$ ,
10
11     // the List module has
12     // a bunch of useful functions
13     List.map(sq, List.range(1, 10))
14 );
15
16 // show result
17 print(sum);
18
```



What next?

- Tuples
- Option Type
- Pattern Matching
- Javascript FFI
- Line Number Error Reporting



Thank You

So long, and thanks for all the fish.

