Functional programming

Lambda Calculus in JavaScript

By Federico Bozzini

TinyJS

TinyJS - the language of the future

Name: TinyJS

Invented in: 2018

Author: Federico Bozzini

TinyJS - the language of the future

Same as javascript but without:

TinyJS - the language of the future

Same as javascript but without:

- with

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans
- numbers

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans
- numbers
- strings

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans
- numbers
- strings
- arrays

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans
- numbers
- strings
- arrays
- operators (>, <, ==, ===, !, &&, ||, +, -, *, /, ...)

TinyJS - the language of the future

Same as javascript but without:

- with
- symbols
- classes
- objects
- exceptions
- booleans
- numbers
- strings
- arrays
- operators (>, <, ==, ===, !, &&, ||, +, -, *, /, ...)
- control structures (if, for, while, ...)

What's left?

What's left?

Functions!!

What’s left?

Functions!!

…yeah…but do to what?

Identity

```
I = x => x
```

```
I(I) ⇒ ???
```

Identity

```
I = x => x
```

```
I(I) ⇒ I
```

First

```
First = (x, y) => x
```

```
First(I, First) ⇒ ???
```

First

```
First = (x, y) => x

First(I, First) ⇒ I
```

Second

```
Second = (x, y) => y

Second(Second, First) ⇒ ???
```

Second

```
Second = (x, y) => y

Second(Second, First) ⇒ First
```

Apply

```
Apply = (f, x) => f(x)

Apply(I, First) ⇒ ???
```

Apply

```
Apply = (f, x) => f(x)

Apply(I, First) ⇒ First
```

ApplyTwice

```
ApplyTwice = (f, x) => f(f(x))

ApplyTwice(I, First) ⇒ ???
```

ApplyTwice

```
ApplyTwice = (f, x) => f(f(x))

ApplyTwice(I, First) ⇒ First
```

TinyJS

Isn’t it cool?

Isn't it cool?

….No

Isn’t it cool?

....Not yet!

TinyJS

What's missing?

TinyJS

What's missing?

Data types!

TinyJS

Is it possible to recreate our data types (Booleans, Numbers, Arrays, Objects, …) using only functions??

Booleans

Booleans

What is a boolean?

Booleans

What is a boolean?

Not a philosophical question!!

```
True, False
Not(True) ⇒ False
And(True, False) ⇒ False
Or(False, True) ⇒ True
```

Booleans

```
True = ???
False = ???
Not = p => ???
And = (p, q) => ???
Or = (p, q) => ???
```

True - False

```
const v = p ? 0 : 1;
```

True - False

```
const v = p ? 0 : 1;

p === true ⇒ 0
p === false ⇒ 1
```

True - False

```
const v = p ? 0 : 1;

p === true ⇒ 0
p === false ⇒ 1
```

True and False must be functions!!

True - False

```
const v = p ? 0 : 1;

p === true ⇒ 0
p === false ⇒ 1
```

True and False must be functions!!

Maybe functions with 2 arguments?

True

```
True = (x, y) => ???
```

True

```
True = (x, y) => x
```

True

```
True = (x, y) => x
```

```
True(I, First) === I
```

True

```
True = (x, y) => x
```

```
True(I, First) === I
```

True ⇔ First !!

False

```
False = (x, y) => ???
```

False

```
False = (x, y) => y
```

False

```
False = (x, y) => y
```

```
False(I, First) === First
```

False

```
False = (x, y) => y
```

```
False(I, First) === First
```

False ⇔ Second !!

Not

```
Not = p => ???
```

Not

P is either True or False

```
Not = p => ???
```

Not

P is either True or False

```
Not = p => p(???, ???)
```

Not

If p is True

```
Not = p => p(???, ???)
```

Not

If p is True

```
Not = p => p(False, ???)
```

Not

If p is False

```
Not = p => p(False, ???)
```

Not

If p is False

```
Not = p => p(False, True)
```

Not

```
Not = p => p(False, True)
```

```
Not(True) ⇒ True(False, True) ⇒ False
```

And

```
And = (p, q) => ???
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

```
And = (p, q) => p(???, ???)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is False

```
And = (p, q) => p(???, ???)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is False

```
And = (p, q) => p(???, ???)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is False

```
And = (p, q) => p(???, ???)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is False

```
And = (p, q) => p(???, p)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is True

```
And = (p, q) => p(???, p)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is True

```
And = (p, q) => p(???, p)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

If p is True

```
And = (p, q) => p(???, p)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

```
And = (p, q) => p(q, p)
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

And

```
And = (p, q) => p(q, p)
```

```
And(True, False)
⇒
True(False, True)
⇒
False
```

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

Or

```
Or = (p, q) => ???
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

```
Or = (p, q) => p(???, ???)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

If p is True

```
Or = (p, q) => p(???, ???)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

If p is True

```
Or = (p, q) => p(p, ???)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

If p is False

```
Or = (p, q) => p(p, ???)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

If p is False

```
Or = (p, q) => p(p, q)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

```
Or = (p, q) => p(p, q)
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Or

```
Or = (p, q) => p(p, q)



Or(False, True)
⇒
False(False, True)
⇒
True
```

p	q	p∨q
T	T	T
T	F	T
F	T	T
F	F	F

Booleans

```
Or(And(True, False), Not(False))
⇒
Or(False, True)
⇒
True
```

If

```
If = (p, x, y) = ???
```

If

```
If = (p, x, y) = p(x, y)
```

If

```
If = (p, x, y) => p(x, y)

If(True, First, Second)
⇒
True(First, Second)
⇒
First
```

Numbers

Numbers (well...natural numbers)

What is a number?

Numbers (well...natural numbers)

What is a number?

583

Numbers (well...natural numbers)

What is a number?

```
583 = 5 * 10^2 + 8 * 10^1 + 3 * 10^0

101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0

IX = 10 - 1
```

Numbers (well...natural numbers)

What is a number?

```
583 = 5 * 10^2 + 8 * 10^1 + 3 * 10^0

101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0

IX = 10 - 1
```

These are number representations

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

```
0       Zero
```

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

```
0        Zero
1        Succ(Zero)
```

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

```
0        Zero
1        Succ(Zero)
2        Succ(Succ(Zero))
```

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

```
0        Zero
1        Succ(Zero)
2        Succ(Succ(Zero))
...
n        Succⁿ(Zero)
```

Natural Numbers

```
Zero = ???
Succ = n => ???
```

Natural Numbers

```
Zero = ???
Succ = n => ???
Add = (m, n) => ???
Mult = (m, n) => ???
Pow = (m, n) => ???
Pred = n => ???
Sub = (m, n) => ???
IsZero = n => ???
Leq = (n, m) => ???
```

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

```
0       Zero
1       Succ(Zero)
2       Succ(Succ(Zero))
...
n       Succⁿ(Zero)
```

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

0	Zero	
1	Succ(Zero)	f(x)
2	Succ(Succ(Zero))	f(f(x))
...		
n	$\text{Succ}^n(\text{Zero})$	$f^n(x)$

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

0	Zero		
1	Succ(Zero)	$f(x)$	Apply
2	Succ(Succ(Zero))	$f(f(x))$	ApplyTwice
...			
n	$\text{Succ}^n(\text{Zero})$	$f^n(x)$	ApplyNTimes

Natural Numbers

Peano axioms, we use only **Zero** and **Succ**:

`0`	`Zero`	`x`	`DontApply`
`1`	`Succ(Zero)`	`f(x)`	`Apply`
`2`	`Succ(Succ(Zero))`	`f(f(x))`	`ApplyTwice`
`...`			
`n`	$\texttt{Succ}^{n}\texttt{(Zero)}$	$\texttt{f}^{n}\texttt{(x)}$	`ApplyNTimes`

Zero

```
Zero = (f, x) => ???
```

Zero

```
Zero = (f, x) => x //don’t apply f to x
```

Zero

```
Zero = (f, x) => x //don't apply f to x
```

`Zero ⇔ False ⇔ Second`

One - Two - n

```
One = (f, x) => f(x) //apply once f to x
```

One - Two - n

```
One = (f, x) => f(x) //apply once f to x
Two = (f, x) => f(f(x)) //apply twice f to x
```

One - Two - n

```
One = (f, x) => f(x) //apply once f to x
Two = (f, x) => f(f(x)) //apply twice f to x
n = (f, x) => fⁿ(x) //apply n time f to x
```

One - Two - n

```
One = (f, x) => f(x) //apply once f to x
Two = (f, x) => f(f(x)) //apply twice f to x
n = (f, x) => fⁿ(x) //apply n time f to x
```

```
One ⇔ Apply
Two ⇔ ApplyTwice
n ⇔ ApplyNTimes
```

Debugging Numbers

```
toInt = n => n(x=>x+1, 0)

toInt(Zero) ⇒ 0
toInt(One) ⇒ 1
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => ???
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x

Clone(One) ⇒
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x

Clone(One) ⇒
(f, x) => One(f, x)
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x

Clone(One) ⇒
(f, x) => One(f, x) ⇒ (f, x) => f(x)
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x

Clone(One) ⇒
(f, x) => One(f, x) ⇒ (f, x) => f(x) ⇔
One
```

Understanding Numbers

```
Zero = (f, x) => x
One = (f, x) => f(x)

Clone = n => (f, x) => n(f, x) //apply n times f to x

Clone(One) ⇒
(f, x) => One(f, x) ⇒ (f, x) => f(x)
One
```

Succ

```
Clone = n => (f, x) => n(f, x) //apply n times f to x
Succ = n => ???
```

Succ

```
Clone = n => (f, x) => n(f, x) //apply n times f to x
Succ = n => (f, x) => f(n(f, x)) //Apply (n+1) times f to x
```

Succ

```
Clone = n => (f, x) => n(f, x) //apply n times f to x
Succ = n => (f, x) => f(n(f, x)) //Apply (n+1) times f to x

Three = Succ(Two)

toInt(Three) ⇒ 3
```

Understanding numbers even better

```
Zero ⇔ Zero(Succ, Zero)
One ⇔ One(Succ, Zero)
Two ⇔ Two(Succ, Zero)
…
n ⇔ n(Succ, Zero)
```

Add

```
Add = (m, n) => ???
```

Add

```
Add = (m, n) => n(Succ, m) //Apply n times f to m
```

Add

```
Add = (m, n) => n(Succ, m) //Apply n times f to m

Four = Add(Three, One) //apply once Succ to Three

toInt(Four) ⇒ 4
```

IsZero

```
IsZero = n => ???
```

IsZero

```
IsZero = n => n(???, ???)
```

IsZero

```
IsZero = n => n(???, True)
```

IsZero

```
IsZero = n => n(x=>False, True)
```

IsZero

```
IsZero = n => n(x=>False, True)

IsZero(Zero) ⇒ True

IsZero(Two) ⇒ False
```

…And more

```
Mult = (m, n) => n(x => Add(x, m), Zero)

Pow = (m, n) => n(x => Mult(x, m), One)

Pred = n => (f, x) => n(g => h => h(g(f)), u=>x) (I)

Sub = (n, m) => n(Pred, m)

Leq = (n, m) => IsZero(Sub(m, n))
```

...And more and more

Division

Negative numbers

Rational Numbers

Real Numbers

Complex numbers

...And more and more and more

Objects

Tuples

Lists

Recursion

...Everything!!

TinyJS - the language of the future

Name: TinyJS

Invented in: 2018

Author: Federico Bozzini

Syntax: (f, x) => f(x)

TinyJS - the language of the ~~future~~ past

Name: Lambda Calculus

Invented In: 1930s

Author: Alonzo Church

Syntax: ƛf.ƛx. f x

(Somehow) First programming language!!

Church Encoding

$$\text{true} \equiv \lambda a.\,\lambda b.\,a$$
$$\text{false} \equiv \lambda a.\,\lambda b.\,b$$
$$\text{and} = \lambda p.\,\lambda q.\,p\;q\;p$$
$$\text{or} = \lambda p.\,\lambda q.\,p\;p\;q$$
$$\text{not}_1 = \lambda p.\,\lambda a.\,\lambda b.\,p\;b\;a \text{ (This is only a correct implementation if the evaluation strategy is applicative order.)}$$
$$\text{not}_2 = \lambda p.\,p\;(\lambda a.\,\lambda b.\,b)\;(\lambda a.\,\lambda b.\,a) = \lambda p.\,p\,\text{false}\,\text{true} \text{ (This is only a correct implementation if the evaluation strategy is normal order.)}$$
$$\text{xor} = \lambda a.\,\lambda b.\,a\;(\text{not}\;b)\;b$$
$$\text{if} = \lambda p.\,\lambda a.\,\lambda b.\,p\;a\;b$$

Table of functions on Church numerals [edit]

Function	Algebra	Identity	Function definition	Lambda expressions	
Successor	$n+1$	$f^{n+1}\;x = f(f^n x)$	$\text{succ}\;n\;f\;x = f\;(n\;f\;x)$	$\lambda n.\,\lambda f.\,\lambda x.\,f\;(n\;f\;x)$	...
Addition	$m+n$	$f^{m+n}\;x = f^m(f^n x)$	$\text{plus}\;m\;n\;f\;x = m\;f\;(n\;f\;x)$	$\lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,m\;f\;(n\;f\;x)$	$\lambda m.\,\lambda n.\,n\,\text{succ}\,m$
Multiplication	$m*n$	$f^{m*n}\;x = (f^m)^n\;x$	$\text{multiply}\;m\;n\;f\;x = m\;(n\;f)\;x$	$\lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,m\;(n\;f)\;x$	$\lambda m.\,\lambda n.\,\lambda f.\,m\;(n\;f)$
Exponentiation	m^n	$n\;m\;f = m^n\;f$[1]	$\exp\;m\;n\;f\;x = (n\;m)\;f\;x$	$\lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,(n\;m)\;f\;x$	$\lambda m.\,\lambda n.\,n\;m$
Predecessor*	$n-1$	$\text{inc}^n\;\text{con} = \text{val}(f^{n-1}x)$	$if(n == 0)\;0\;else\;(n-1)$	$\lambda n.\,\lambda f.\,\lambda x.\,n\;(\lambda g.\,\lambda h.\,h\;(g\;f))\;(\lambda u.\,x)\;(\lambda u.\,u)$	
Subtraction*	$m-n$	$f^{m-n}\;x = (f^{-1})^n(f^m x)$	$\text{minus}\;m\;n = (n\,\text{pred})\;m$	...	$\lambda m.\,\lambda n.\,n\,\text{pred}\,m$

References

https://en.wikipedia.org/wiki/Lambda_calculus

https://en.wikipedia.org/wiki/Church_encoding

https://www.jtolio.com/2017/03/whiteboard-problems-in-pure-lambda-calculus/

https://www.youtube.com/watch?v=3VQ382QG-y4

https://tadeuzagallo.com/blog/writing-a-lambda-calculus-interpreter-in-javascript/

Fin!

Comments? Questions?