|>

Lisp w JavaScript

Marcin Najder - **Comarch**

@marcinnajder

Lisp w JavaScript

- Ile jest Lisp w JavaScript
- Interpreter Lisp w JavaScript
  - .. w podejściu funkcyjnym w TypeScript

Historia JavaScript

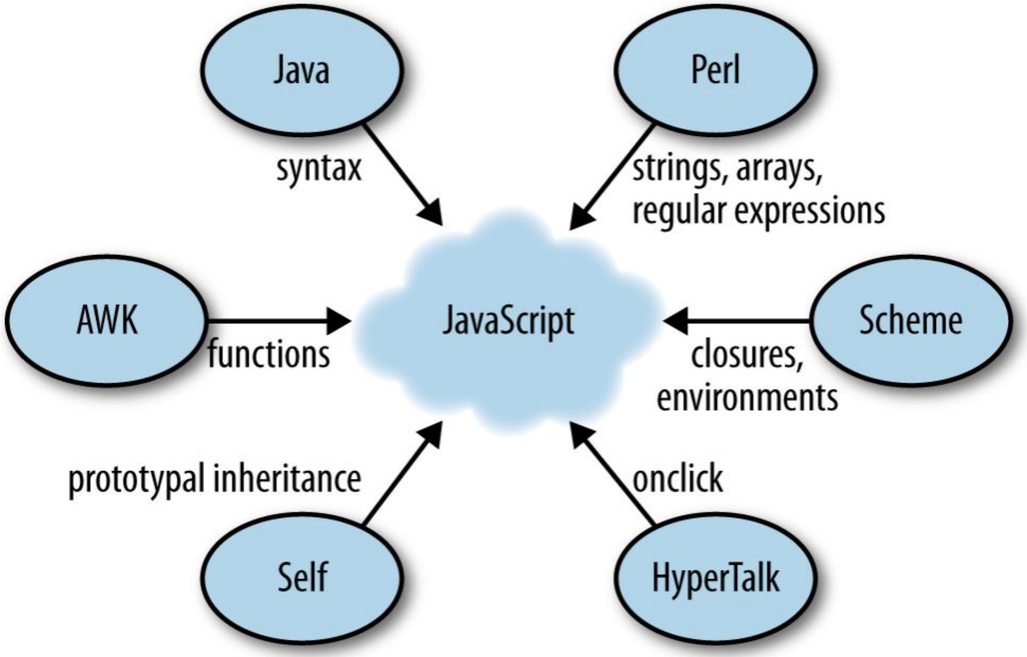


Figure 3-1. Programming languages that influenced JavaScript.

http://speakingjs.com/es5/ch03.html

Funkcyjne języki programowania

rodzina języków "Lisp"
- 1958 – Lisp
- 1970 - Scheme
- 1981 - Common Lisp
- 1994 - Racket
- 2007 - Clojure (ClojureScript)

rodzina języków "ML"
- 1973 – ML
- 1984 - SML
- 1996 - OCaml
- 2005 - F# (Fable)
- 2016 - ReasonML

- 1990 – Haskell (PureScript, Elm)

mal - Make a Lisp

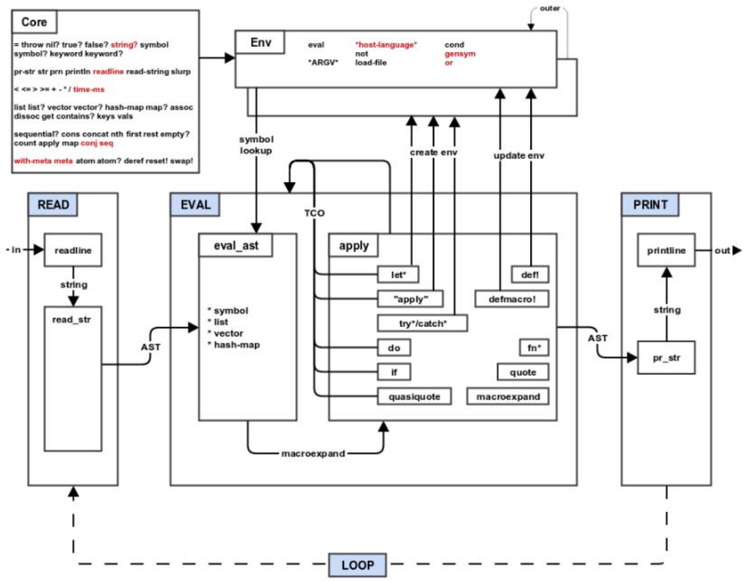
https://github.com/kanaka/mal/

build passing

Description

1. Mal is a Clojure inspired Lisp interpreter

2. Mal is implemented in 79 languages (81 different implementations and 102 runtime modes)



Core
= throw nil? true? false? string? symbol symbol? keyword keyword?
pr-str str prn println readline read-string slurp
< <= > >= + - * / time-ms
list list? vector vector? hash-map map? assoc dissoc get contains? keys vals
sequential? cons concat nth first rest empty? count apply map conj seq
with-meta meta atom atom? deref reset! swap!
outer
Env
eval
ARGV
host-language
not
load-file
cond
gensym
or
symbol lookup
create env
update env
TCO
READ
- in
readline
string
read_str
AST
EVAL
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
try*/catch*
do
if
quasiquote
def!
defmacro!
fn*
quote
macroexpand
macroexpand
AST
PRINT
printline
out
string
pr_str
LOOP

The Make-A-Lisp Process

- Step 0: The REPL
- Step 1: Read and Print
- Step 2: Eval
- Step 3: Environments
- Step 4: If Fn Do
- Step 5: Tail call optimization
- Step 6: Files, Mutation, and Evil
- Step 7: Quoting
- Step 8: Macros
- Step 9: Try
- Step A: Metadata, Self-hosting and Interop

Lisp - "LISt Processor"

```
(+ 1 2)                                            ; -> 3

(/ (* 2 3) (+ 1 2))                                ; -> 2

(< 10 15)                                          ; -> true

(if (< 10 15) (+ 1 2) 5)                           ; -> 3
```

Environment

```
(def! a 1)                                        ; -> 1

(def! b (+ 1 2 3 ))                               ; -> 6

(if (= a b) 10 1000)                              ; -> 1000

(let* (x 100 y 10) (/ x y))                       ; -> 10

(let* (x 100 y (* 5 x)) (/ x y))                  ; -> 0.2
```

Typy danych

```
(let* (s "text" b true n 5.5 ni nil) n )  ; -> 5.5


(string? "text")                          ; -> true
(number? 234)                             ; -> true
(nil? nil)                                ; -> true
(true? true)                              ; -> true
(false? false)                            ; -> true


(list? '(1 2 ))                           ; -> true
(vector? [1 2])                           ; -> true
(map? {})                                 ; -> true

(fn? +)                                   ; -> true
(symbol? 's)                              ; -> true
(keyword? :k)                             ; -> true
```

Listy, wektory

```
(def! list1 (list 10 40 100))                ; -> (10 40 100)


(def! list2 (list "text" true 5.5 nil))      ; -> ("text" true 5.5 nil)


(def! vector1 (vector 10 40 100))            ; -> [10 40 100]
(def! vector1 [10 40 100])                   ; -> [10 40 100]


(conj (list 6 7 8) 1 2 3)                    ; -> (3 2 1 6 7 8)
(conj [6 7 8] 1 2 3)                         ; -> [6 7 8 1 2 3]
```

Listy, wektory

```
(def! list1 (list 10 40 100))                     ; -> (10 40 100)

(def! list2 (quote (10 40 100)))                  ; -> (10 40 100)
(def! list2 `(10 40 100))                         ; -> (10 40 100)


(cons 0 '(1 2 ))                                  ; -> (0 1 2)

(first '(8 7 5))                                  ; -> 8
(rest '(8 7 5))                                   ; -> (7 5)

(count '(0 1 2))                                  ; -> 3
(empty? '())                                      ; -> true
(concat '(1 2 ) '(7 8 9))                         ; -> (1 2 7 8 9)
(nth '(8 7 5) 1)                                  ; -> 7
```

Mapy

```
(def! map1 {:name "John" :age 30})                  ; -> {:name "John" :age 30}
(def! map1 (hash-map :name "John":age 30) )         ; -> {:name "John" :age 30}

(get map1 :name)                                    ; -> "John"
(get map1 :city)                                    ; -> nil
(contains? map1 :age)                               ; -> true
(keys map1)                                         ; -> (:name :age)
(vals map1)                                         ; -> ("John" 30)


(def! map2 (assoc map1 :city "L.A."))   ; -> {:name "John" :age 30 :city "L.A."}
(def! map3 (dissoc map2 :city ))                    ; -> {:name "John" :age 30}


(= map1 map3)                                       ; -> true
(= '(1 2 3) (cons 1 (list 2 3)))                    ; -> true
```

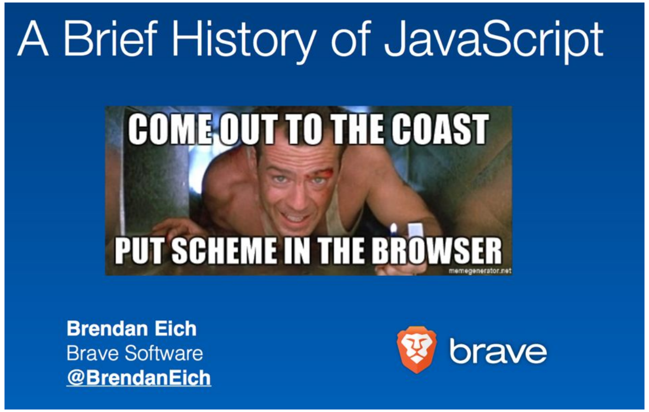
Funkcje, closure

```
(def! inc (fn* (x) (+ x 1)))                   ; -> #<function>
(inc 10)                                       ; -> 11


(def! add (fn* (a b) (+ a b)))                 ; -> #<function>

(def! max (fn* (a b) (if (> a b) a b)))        ; -> #<function>

(def! hello-world (fn* ()
  (do
    (prn "hello")
    (prn "world") )))


; closure
(def! return-value (fn* (value) (fn* () value)))
(def! return-10 (return-value 10))

(return-10)                                    ; -> 10
```

Funcja ‘map’

```
(def! map (fn* (f xs)
  (if (empty? xs)
    xs
    (cons (f (first xs)) (map f (rest xs))))))



(map inc `(1 2 3) )                                    ; -> (2 3 4)


(map (fn* (x) (* x 10)) `(1 2 3) )                     ; -> (10 20 30)
```

Czy JavaScript to Lisp?

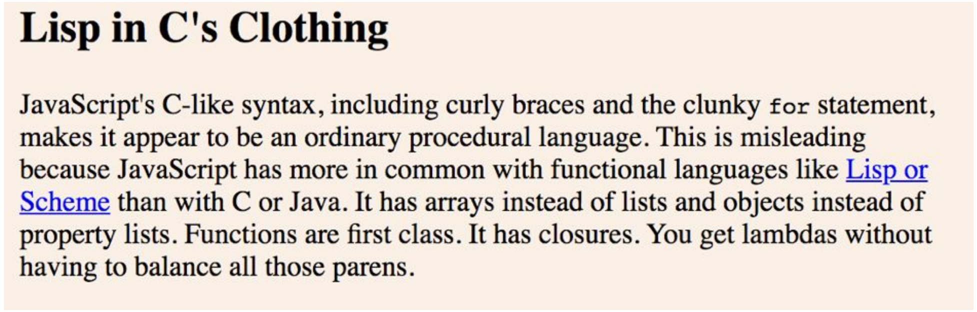


A Brief History of JavaScript

Brendan Eich
Brave Software
@BrendanEich

brave

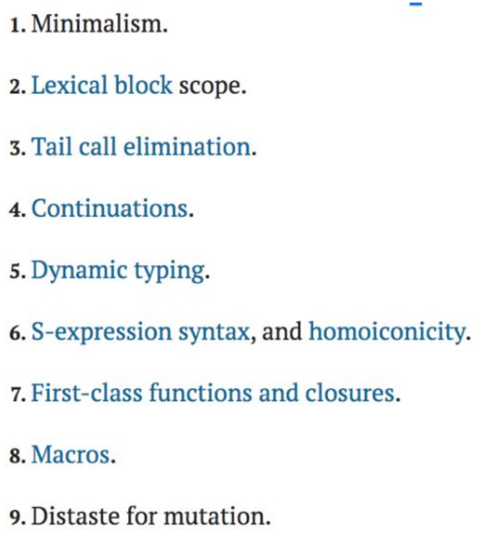
https://brendaneich.com/2017/12/my-dotjs-2017-keynote/



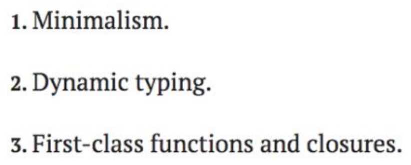
Lisp in C's Clothing

JavaScript's C-like syntax, including curly braces and the clunky `for` statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java. It has arrays instead of lists and objects instead of property lists. Functions are first class. It has closures. You get lambdas without having to balance all those parens.

https://www.crockford.com/javascript/javascript.html

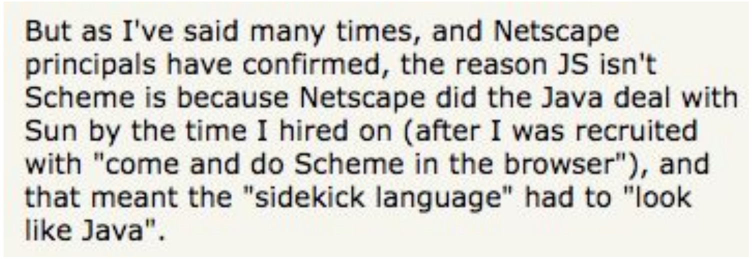


1. Minimalism.
2. Lexical block scope.
3. Tail call elimination.
4. Continuations.
5. Dynamic typing.
6. S-expression syntax, and homoiconicity.
7. First-class functions and closures.
8. Macros.
9. Distaste for mutation.



1. Minimalism.
2. Dynamic typing.
3. First-class functions and closures.

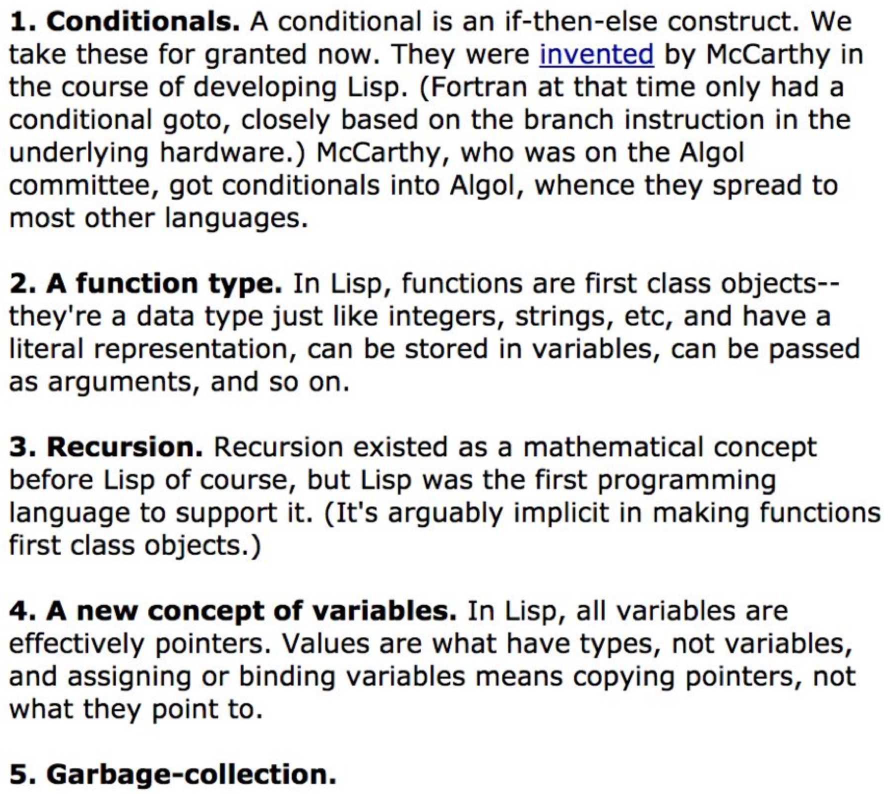
http://journal.stuffwithstuff.com/2013/07/18/javascript-isnt-scheme/



But as I've said many times, and Netscape principals have confirmed, the reason JS isn't Scheme is because Netscape did the Java deal with Sun by the time I hired on (after I was recruited with "come and do Scheme in the browser"), and that meant the "sidekick language" had to "look like Java".

https://news.ycombinator.com/item?id=13142737

Innowacje Lisp



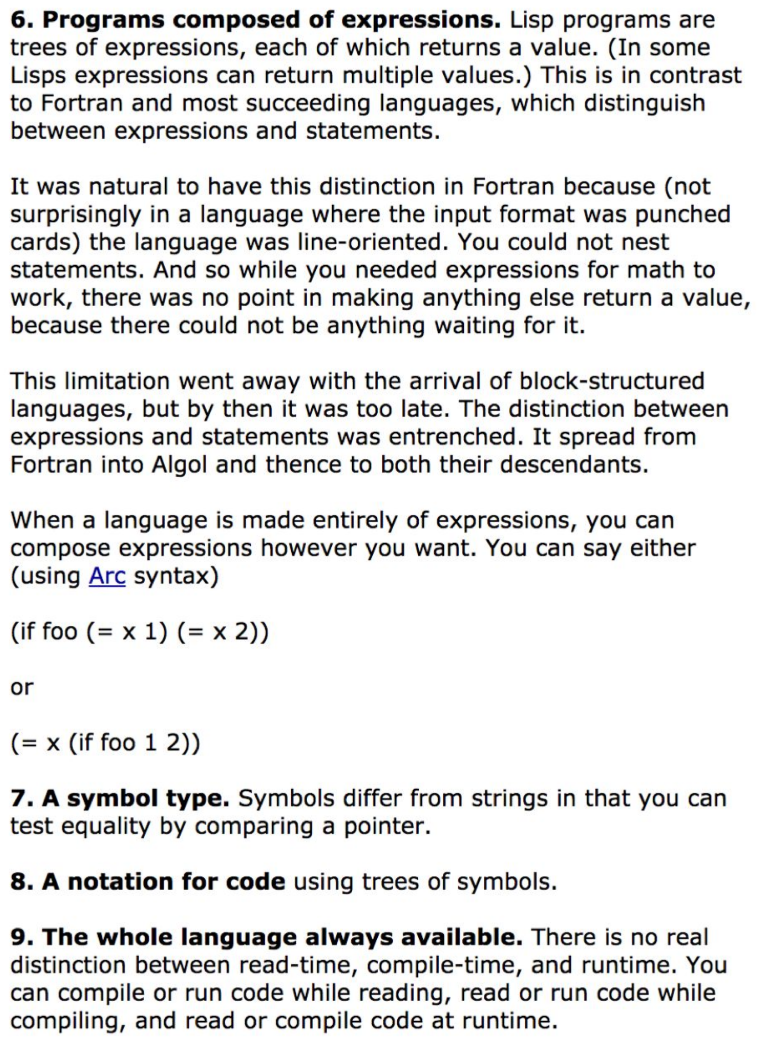
1. Conditionals. A conditional is an if-then-else construct. We take these for granted now. They were invented by McCarthy in the course of developing Lisp. (Fortran at that time only had a conditional goto, closely based on the branch instruction in the underlying hardware.) McCarthy, who was on the Algol committee, got conditionals into Algol, whence they spread to most other languages.

2. A function type. In Lisp, functions are first class objects-- they're a data type just like integers, strings, etc, and have a literal representation, can be stored in variables, can be passed as arguments, and so on.

3. Recursion. Recursion existed as a mathematical concept before Lisp of course, but Lisp was the first programming language to support it. (It's arguably implicit in making functions first class objects.)

4. A new concept of variables. In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.

5. Garbage-collection.



6. Programs composed of expressions. Lisp programs are trees of expressions, each of which returns a value. (In some Lisps expressions can return multiple values.) This is in contrast to Fortran and most succeeding languages, which distinguish between expressions and statements.

It was natural to have this distinction in Fortran because (not surprisingly in a language where the input format was punched cards) the language was line-oriented. You could not nest statements. And so while you needed expressions for math to work, there was no point in making anything else return a value, because there could not be anything waiting for it.

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from Fortran into Algol and thence to both their descendants.

When a language is made entirely of expressions, you can compose expressions however you want. You can say either (using Arc syntax)

```
(if foo (= x 1) (= x 2))
```

or

```
(= x (if foo 1 2))
```

7. A symbol type. Symbols differ from strings in that you can test equality by comparing a pointer.

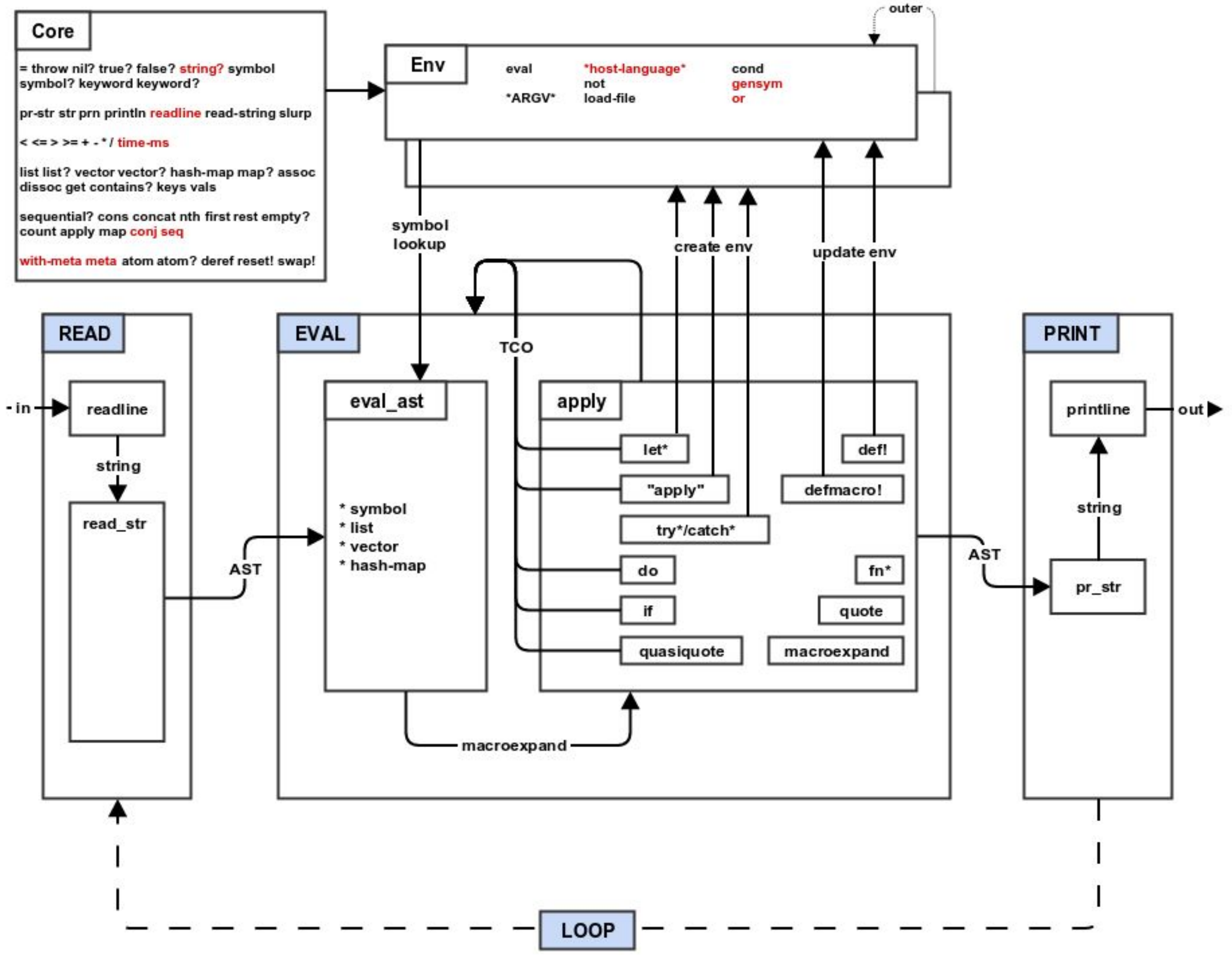
8. A notation for code using trees of symbols.

9. The whole language always available. There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

http://www.paulgraham.com/diff.html

Interpreter Lisp w TypeScript

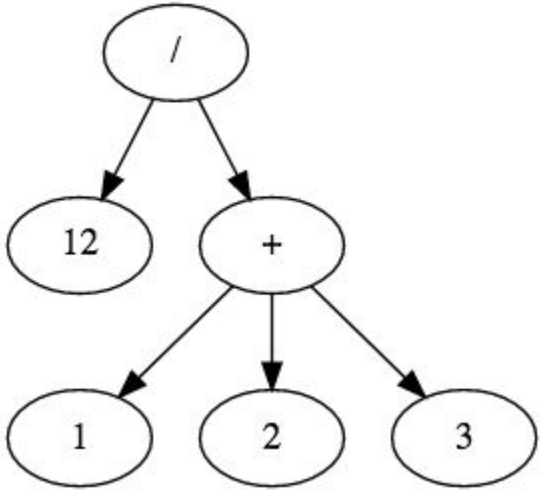
REPL

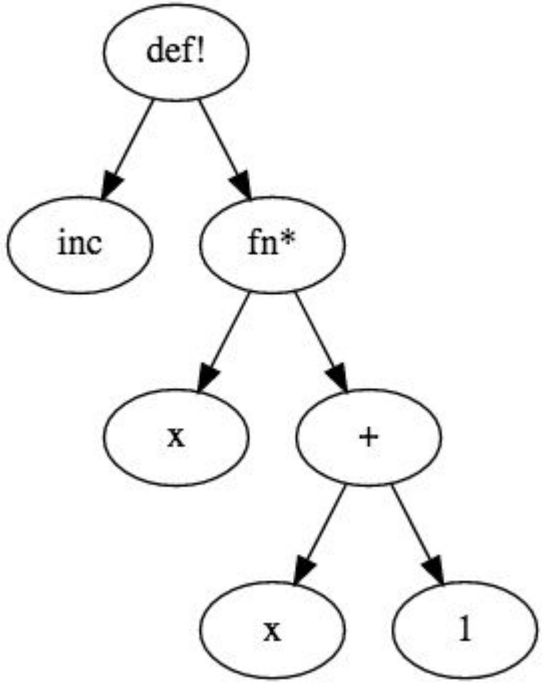

Core
= throw nil? true? false? string? symbol symbol? keyword keyword?
pr-str str prn println readline read-string slurp
< <= > >= + - * / time-ms
list list? vector vector? hash-map map? assoc dissoc get contains? keys vals
sequential? cons concat nth first rest empty? count apply map conj seq
with-meta meta atom atom? deref reset! swap!
Env
eval
ARGV
host-language
not
load-file
cond
gensym
or
outer
symbol lookup
create env
update env
TCO
READ
- in
readline
string
read_str
AST
EVAL
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
try*/catch*
do
if
quasiquote
def!
defmacro!
fn*
quote
macroexpand
macroexpand
AST
PRINT
printline
out
string
pr_str
LOOP

Abstract Syntax Tree (AST)

```
(/ 12 (+ 1 2 3))
```

```
(def! inc (fn* (x) (+ x 1)))
```





AST obiektowo

```
interface Expression {
  print(): string;
  eval_(): Expression;
  // ...
}

class Number_ extends Expression {
    constructor(public value: number) { }
    print() { return this.value.toString() }
}
class Symbol_ extends Expression {
    constructor(public name: string) { }
    print() { return this.name.toString() }
}
class List extends Expression {
    constructor(public items: Expression[]) { }
    print() { return `(${this.items.map(e => e.print()).join(" ")})`; }
}


const e = new List([ new Symbol_("+"), new Number_(1), new Number_(2) ]);
console.log(e.print());    // -> (+ 1 2)
```

AST funkcyjnie

```
class Number_  {
    constructor(public value: number) { }
}
class Symbol_  {
    constructor(public name: string) {  }
}
class List  {
    constructor(public items: any[]) {  }
}

function print(expr: any): string {
    if (expr instanceof Number_) { return expr.value.toString(); }
    if (expr instanceof Symbol_) { return expr.name; }
    if (expr instanceof List) { return `(${expr.items.map(print).join(" ")})`; }
    throw new Error("Unknown expression");
}

const e = new List([ new Symbol_("+"), new Number_(1),new Number_(2)]);
console.log(print(e));    // -> (+ 1 2)

// Które podejście lepsze ?
// https://en.wikipedia.org/wiki/Expression_problem - Expression Problem
```

Algebraiczne typy danych

Optional<T>

```
interface Optional<T>{
    hasValue: boolean;
    value?: T;
}

function tryParseNumber(text: string): Optional<number> {
    const n = Number(text);
    return Number.isNaN(n) ? { hasValue: false } : { hasValue: false, value: n};
}


// bledne stany !!
const o1: Optional<number> = { hasValue: false, value: 123 };
const o2: Optional<number> = { hasValue: true };



// https://www.youtube.com/watch?v=IcgmSRJHu_8 -
// "Making Impossible States Impossible" by Richard Feldman
```

Option<T>

```
// Maybe = Just | Nothing
type Option<T> =
    | { type: "some", value: T }
    | { type: "none" };


function tryParseNumber(text: string): Option<number> {
  const n = Number(text);
  return Number.isNaN(n) ? { type: "none"} : {type: "some", value: n}
}



// bledy kompilacji TS :)
const o1: Option<number> = { type: "none", value: 123 };
const o2: Option<number> = { type: "some" };
```

Algebraic data types (ADT)

```
// * product/each_of/AND type
interface Type1 {
    item1: boolean;
    item2: byte;
}
// count(Type1) = count(boolean) * count(byte) = 2 * 256 = 512



// + sum/one_of/OR/choice type
type Type2 = boolean | byte;
// count(Type2) = count(boolean) + count(byte) = 2 + 256 = 258
```

Konstruktory typów

```
type Option<T> =
    | { type: "some", value: T }
    | { type: "none" };


function some<T>(value: T): Option<T> {
  return { type: "some", value };
}
function none<T>(): Option<T> {
  return { type: "none" };
}


function tryParseNumber(text: string): Option<number> {
  const n = Number(text);
  return Number.isNaN(n) ? none() : some(n);
}
```

Option<T> w powerfp

```
import { Option, none, some, Option_some, Option_none } from "powerfp";

function tryParseNumber(text: string): Option<number> {
  const n = Number(text);
  return Number.isNaN(n) ? none : some(n);
}


// ** powerfp **
type Option<T> =
  | { type: "some", value: T }
  | { type: "none" };

// kod automatycznie wygenerowany za pomoca biblioteki 'powerfp'
const some = <T>(value: T) => ({ type: "some", value }) as Option_some<T>;
const none = { type: "none" } as Option_none;

type Option_some<T> = { type: "some", value: T };
type Option_none = { type: "none" };
```

Pattern matching

```
import { Option } from "powerfp";

function format(n: Option<number>): string{
    switch (n.type) {
        case "some": return `${Math.round(n.value)} zl`;
        case "none": return "0 zl";
    }
}

// https://www.typescriptlang.org/docs/handbook/advanced-types.html#exhaustiveness-checking
```

Pattern matching w powerseq

```
import { matchUnion, isUnion } from "powerfp";


function format(n: Option<number>) : string{
    return matchUnion(n, {
        some: ({ value }) => `${Math.round(value)} zl`,
        none: () => "0 zl"
    });
}



const o: Option<number> = ...;
console.log(isUnion(o, "some") ? o.value : "");
```

Option<T> jest Monad ...

```
import { optionMapM } from "powerfp";

function console_tryReadNumber(): Option<number> { ... }


console_tryReadNumber().map(a => Math.round(a));        // -> Option<number>

console_tryReadNumber().bind( a => console_tryReadNumber().map(b => a + b));




// funkcje mapM, filterM, reduceM, ...
["1", "2.6", "a"].map(v => tryParseNumber(v));          // -> Array<Option<number>>

optionMapM(["1", "2.6", "a"], v => tryParseNumber(v));  // -> Option<Array<number>>




// https://vimeo.com/343153316 - Monady w JavaScript
```

Result<T, E> i ResultS<T>

```
// Either = Left | Right
type Result<T, E> =
    | { type: "ok", value: T }
    | { type: "error", error: E };

type ResultS<T> = Result<T, string>;
```

Result<T, E> i ResultS<T> w powerfp

```
import { Result, error, ok, matchUnion, Option } from  "powerfp";


function tryParseIntNumber(text: string): Result<number, string> {
  const n: Option<number> = tryParseNumber(text);

  return matchUnion(n, {
    none: () => error("text is not a number"),
    some: ({ value }) => Number.isInteger(value) ?
      ok(value) : error("text is not an integer")
  });
}
```

Interpreter Lisp w TypeScript

AST dla Mal

```
type MalType =
  | { type: "number_"; value: number }
  | { type: "symbol"; name: string }
  | { type: "list"; items: MalType[]; listType: "list"|"vector" }
  | { type: "nil" }
  | { type: "true_" }
  | { type: "false_" }
  | { type: "string_"; value: string }
  | { type: "keyword"; name: string }
  | { type: "fn"; fn: (args: MalType[]) => ResultS<MalType>) }
  | { type: "atom"; mal: MalType }
  | { type: "map"; map: { [key:  string]:  MalType } };


// kod wygenerowany za pomoca powerfp
const number_ = (value: number) => ({ type: "number_", value }) as MalType_number_;
type MalType_number_ = UnionChoice<MalType, "number_">;
// ...

// https://github.com/marcinnajder/misc/blob/master/2019_04_06_make_a_lisp_in_typescript/src/types.ts
```

Read Evaluate Print Loop (REPL)

```
while (true) {
  const inputText = readLine();                // "(+ 1 2)"
  const tokens = tokenize(inputText);          // ["(", "+", "1", "2", ")"]
  const inputMal = READ(tokens);

  // { type: "list", listType: "list", items: [
  //     { type: "symbol", name: "+" },
  //     { type: "number_", value: 1 },
  //     { type: "number_", value: 2 } ] }

  const outputMal = EVALUATE(inputMal);        // { type: "number_", value: 3 }
  const outputText = PRINT(outputMal);         //  3
  writeLine(outputText);
}
```

print

```
function print(mal: MalType): string {
  return matchUnion(mal, {
    number_: ({ value }) => value.toString(),
    symbol: ({ name }) => name,
    string_: ({ value }) => `"${value}"`,
    keyword: ({ name }) => `:${name}`,
    true_: () => "true",
    false_: () => "false",
    nil: () => "nil",
    list: ({ listType, items }) => {
      const b = { "list": ["(", ")"], "vector": ["[", "]"] };
      return `${b[listType][0]}${items.map(print).join(" ")}${b[listType][0]}`;
    },
    fn: _ => "#<function>",
    // ...
  });
}
```

tokenize

```
/[\s,]*(~@|[\[\]{}()'`~^@]|"(?:\\.|[^\\"])*"?|;.*|[^\s\[\]{}('"`,;)]*)/g;


// regex rozpoznaje nastepujace sekwencje znakow
// -> ~@ [ ] { } ( ) ' ` ~ ^@
// -> "..."
// -> abc :abc 123 true false nil



// (def! p {:id 1 :name "John"})

// -> [ '(', 'def!', 'p', '{', ':id', '1', ':name', '"John"', '}', ')' ]
```

read

```
interface Reader {
  next(): Option<string>;
  peek(): string;
}

function read_form(reader: Reader): ResultS<Option<MalType>> {
    const t = reader.peek();
    // ...
}

// ['"']                         -> error(`String value '"' in not closed`)
// ['']                          -> ok(none)
// ['5']                         -> ok(some(number_(5)))
// ['(', '+', '1', '2', ')']  ->
                    ok(some(list([symbol("+"),number_(1),number_(2)],"list")))

//https://github.com/marcinnajder/misc/blob/master/2019_04_06_make_a_lisp_in_typescript/src/reader.ts
```

Environment

```
interface Env {
  parent: Option<Env>;        // ~proto

  set(key: string, mal: MalType): MalType;
  get(key: string): ResultS<MalType>;
}


// > (def! n 10)
env.set("n", number_(10));

// > (+ n 100)
eval_( '(+ n 100), env);


// > (let* (a 10 b (+ a 100)) b)
env1 = new Env(some(env));
env1.set("a", number_(10));

env2 = new Env(some(env1));
env2.set("b", eval_( '(+ a 100), env1 ));
eval_( 'b, env2 );
```

fn*

```
// type MalType =
//  | ...
//  | { type: "fn"; fn: (args: MalType[]) => ResultS<MalType>; };


// > (def! inc (fn* (x) (+ x 1)))
env.set("inc", fn( (args: MalType[]) => {
    const env1 = new Env(env);
    env1.set("x", args[0]);
    return eval_( '(+ x 1), env1);
}));



// > (inc 10)
eval_( '(inc 10) , env);
```

core

```
// predefiniowany zestaw funkcji +,*,-,/,=,<=,>,>=,list,list?,empty, ...

env.set("+", fn( (args: MalType[]) => {
  const sum = (args as MalType_number_[]).reduce((s, mal) => s + mal.value, 0);
  return ok(number_(sum));
}));


env.set("<", fn( (args: MalType[]) => {
  const [first, second] = args as MalType_number_[];
  return ok(first.value < second.value ? true_ : false_));
}));



//https://github.com/marcinnajder/misc/blob/master/2019_04_06_make_a_lisp_in_typescript/src/core.ts
```

eval_

```
function eval_(mal: MalType, env: Env): ResultS<MalType> {
  mal = macroexpand(mal, env);

  if(isUnion(mal, "list") && mal.listType === "list"){
    return apply_list(mal, env);
  }

  return eval_ast(mal, env);
}


function eval_ast(mal: MalType, env: Env): ResultS<MalType> {
  return matchUnion(mal, {
    symbol: ({ name }) => env.get(name),
    list: ({ items, listType }) => resultSMapM(items, m => eval_(m, env))
                  .map(evaluatedItems => list(evaluatedItems, listType, nil)),
    map: mal =>  ...
    _: () => ok(mal)
  });
}
```

apply_list

```
function apply_list(mal: MalType_list, env: Env): ResultS<MalType> {
  const { items: [symbol, ...args] } = mal;

  switch ((symbol as MalType_symbol).name) {
    case "def!": {
      const [sym, mal] = args as [MalType_symbol, MalType];
      return eval_(mal, env).map(m => env.set(sym.name, m));
    }
    case "fn*": {
      const [{ items: fnArgs }, fnBody] = args as [MalType_list, MalType];
      return ok(fn((fnCallArgs) => {
        const env1 = new Env(some(env));
        (fnArgs as MalType_symbol[]).forEach(({ name }, i) => env1.set(name, fnCallArgs[i]));
        return eval_(fnBody, env1);
      }, nil));
    }
    // case "if", "let*", "do", "quote", "try*" ...

    default: {
      return eval_ast(mal, env).bind(m => {
        const { items: [fn, ...fnArgs] } = m as MalType_list;
        return (fn as MalType_fn).fn(fnArgs);
      });
    }
  }
}
```

Przykładowa wykonanie

```
> (def! inc (fn* (x) (+ x 1)))
> (inc 10)
```

```
// (def! inc (fn* (x) (+ x 1)))


{ type: 'list', listType: 'list', items: [
  { type: 'symbol', name: 'def!' },
  { type: 'symbol', name: 'inc' },
  { type: 'list', listType: 'list', items: [
    { type: 'symbol', name: 'fn*' },
    { type: 'list', listType: 'list', items: [
      { type: 'symbol', name: 'x' }]
    },
    { type: 'list', listType: 'list', items: [
      { type: 'symbol', name: '+' },
      { type: 'symbol', name: 'x' },
      { type: 'number_', value: 1 }]
    }]
  }]
}
```

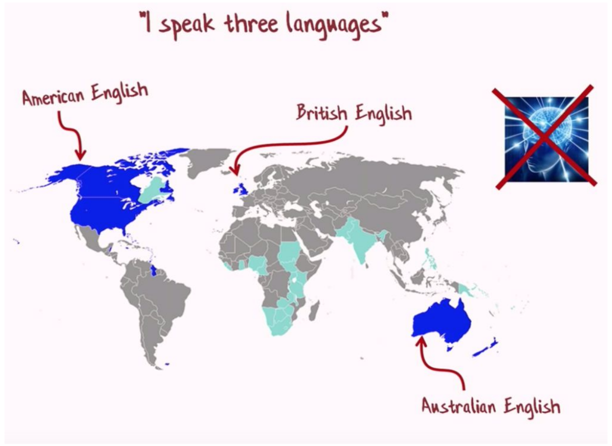
```
> (def! inc (fn* (x) (+ x 1)))

eval_( '(def! inc (fn* (x) (+ x 1))) , env);


env["inc"] = eval_( '(fn* (x) (+ x 1)), env)

env["inc"] = { type: "fn", fn: (args) => _eval( '(+ x 1), { x: args[0] } ) }
```

```
> (inc 10)

eval_( '(inc 10), env )
eval_( 'inc , env)
eval_( '10 , env)


{ type: list, listType: 'list', items: [
  { type: "fn", fn: (args) => _eval('(+ x 1), { x: args[0] })  },
  { type: "number_", value: 10} ]}


( (args) => _eval(`(+ x 1)`, { x: args[0] }) ) ( '10 )

_eval('(+ x 1), { x: '10 })
_eval('(+ x 1), { x: '10 })
eval_( '+ , { x: '10 })
eval_( 'x , { x: '10 })
eval_( '1 , { x: '10 })


{ type: list, listType: 'list', items: [
  { type: "fn", fn: (args) => {
      const sum = (args as MalType_number_[]).reduce((s, mal) => s + mal.value, 0);
      return ok(number_(sum));
    },
  { type: "number_", value: 10}
  { type: "number_", value: 1} ]}


( (args) => ... ) (['10, '1])

{ type: "number_", value: 11}
```

demo

Podsumowanie

- Lisp może nas wiele nauczyć
- JavaScript = Lisp, TypeScript = ML ?
- Zadanie domowe
  - https://fsharpforfunandprofit.com/ddd/ Domain Driven Design with the F# type System
- Warto się uczyć języków …



It's a big world out there
Not every language looks like C/C#/Java/JavaScript

https://youtu.be/0fpDlAEQio4?t=720 Four Languages from Forty Years Ago - Scott Wlaschin

Materiały

- Lisp
  - https://mitpress.mit.edu/sites/default/files/sicp/index.html Structure and Interpretation of Computer Programs
  - http://www.paulgraham.com/onlisp.html On Lisp
  - https://www.crockford.com/little.html The Little JavaScripter
  - http://www.paulgraham.com/lisp.html lisp
- SML/OCaml, LISP, Ruby
  - https://courses.cs.washington.edu/courses/cse341/13wi/ Programming Languages
- Kod
  - https://github.com/kanaka/mal/ Make a lisp
  - https://github.com/marcinnajder/misc/tree/master/2019_04_06_make_a_lisp_in_typescript%20Lisp%20w%20TypeScript Mal w TypeScript
  - https://github.com/marcinnajder/powerfp powerfp
  - https://github.com/marcinnajder/powerseq powerseq

- https://www.youtube.com/watch?v=3-9fnjzmXWA A Brief History of JavaScript by the Creator of JavaScript

Dziękuję za uwagę !