REGURSION

STRUCTURE OF THE TALK

- » Recursion
- » Corecursion
- » Recursion schemes aka. higher level magic

SUMMING NUMBERS RECURSIVELY

```
// range(0, 1000)
let numbers: number[] = Array.from(Array(1000).keys());
function sumRecur(xs: number[]): number {
  if (xs.length < 1) {</pre>
    return 0;
  return xs[0] + sumRecur(xs.slice(1, xs.length));
console.log(sumRecur(numbers));
// 499500
```

HOW DOES THIS EVALUATE?

```
let nums = [0, 1, 2, 3]
sumRecur(nums)
0 + sumRecur([1, 2, 3])
0 + (1 + sumRecur([2, 3]))
0 + (1 + (2 + sumRecur([3])))
0 + (1 + (2 + (3 + sumRecur([]))))
0 + (1 + (2 + (3 + 0)))
0 + (1 + (2 + 3))
0 + (1 + 5)
0 + 6
6
```

TAIL RECURSION

"Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations" Wikipedia

TAIL-RECURSIVE SUM

```
function sumTailRecur(nums: number[], acc: number): number {
  if (nums.length < 1) {</pre>
    return acc
  return sumTailRecur(nums.slice(1, nums.length), acc + nums[0])
console.log(sumTailRecur(numbers, 0));
// 499500
```

TAIL-RECURSIVE SUM

```
function sumTailRecur(xs: number[]): number {
  function sumR(nums: number[], acc: number): number {
    if (nums.length < 1) {</pre>
      return acc;
    return sumR(nums.slice(1, nums.length), acc + nums[0])
 return sumR(xs, 0);
console.log(sumTailRecur(numbers));
  499500
```

HOW DOES THIS EVALUATE?

```
let nums = [0, 1, 2, 3]
sumTailRecur(nums)
sumR([0, 1, 2, 3], 0)
sumR([1, 2, 3], 0)
sumR([2, 3], 1)
sumR([3], 3)
sumR([], 6)
```

NOTICE THE REPITITION

```
function sumTailRecur(xs: number[]): number {
 function sumR(nums: number[], acc: number): number {
    if (nums.length < 1) {</pre>
      return acc;
    return sumR(
        nums.slice(1, nums.length),
        acc + nums[0]
   );
  return sumR(
   XS,
   0
  );
function lengthTailRecur(xs: number[]): number {
  function lengthR(nums: number[], acc: number): number {
   if (nums.length < 1) {</pre>
     return acc;
    return lengthR(
       nums.slice(1, nums.length),
       acc + 1
   );
  return lengthR(
   xs,
   0
  );
```

```
function maxTailRecur(xs: number[]): number {
 function maxR(nums: number[], acc: number): number {
    if (nums.length < 1) {</pre>
      return acc;
   return maxR(
        nums.slice(1, nums.length),
       acc > nums[0] ? acc : nums[0]
   );
  return maxR(
   xs,
   Number.MIN_VALUE
 );
function minTailRecur(xs: number[]): number {
 function maxR(nums: number[], acc: number): number {
   if (nums.length < 1) {</pre>
      return acc;
    return maxR(
        nums.slice(1, nums.length),
        acc < nums[0] ? acc : nums[0]
   );
  return maxR(
   xs,
   Number.MAX_VALUE
  );
```

```
function recurser(
    xs: number[],
    f: (x: number, acc: number) => number,
    base: number
): number {
      function helper(nums: number[], acc: number): number {
        if (nums.length < 1) {</pre>
          return acc;
        return helper(nums.slice(1, nums.length), f(acc, nums[0]));
      return helper(xs, base);
```

FOLDS BEAUTIFUL FOLDS AKA. REDUCE

```
function foldLeft(
    xs: number[],
    f: (acc: number, x: number) => number,
    base: number
): number {
      function helper(nums: number[], acc: number): number {
        if (nums.length < 1) {</pre>
          return acc;
        return helper(nums.slice(1, nums.length), f(acc, nums[0]));
      return helper(xs, base);
```

John Hughes (1990) Why Functional Programming Matters. Research Topics in Functional Programming.

```
if (nums.length < 1) {</pre>
      nums.slice(1, nums.length),
      acc + nums[0]
 0
 return lengthR(
     acc + 1
return lengthR(
 0
```

```
nums.slice(1, nums.length),
    acc > nums[0] ? acc : nums[0]
Number.MIN_VALUE
if (nums.length < 1) {
    nums.slice(1, nums.length),
    acc < nums[0] ? acc : nums[0]
Number.MAX_VALUE
```




su (Average numerator denominator) = numerator / fromIntegral denominator

CORECURSION

IF RECURSION WAS YIN, CORECURSION WOULD BE YANG

CORECURSION

"corecursion is a type of operation that is dual to recursion"
Wikipedia

- » Recursion starts with data and works (or folds) down towards the base case.
- » Corecursion is able use data that it produces from the base case.

EXAMPLES

This is where the wheels will probably fall off

RECURSION VS. CORECURSION COMPARISONS

https://en.wikipedia.org/wiki/Corecursion#Examples

RECURSION SCHEMES

Meijer E., Fokkinga M., Paterson R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes J. (eds) Functional Programming Languages and Computer Architecture. FPCA 1991. Lecture Notes in Computer Science, vol 523. Springer, Berlin, Heidelberg

Recursion Schemes

folds (tear down a structure) unfolds (build up a structure) algebra $f a \rightarrow Fix f \rightarrow a$ coalgebra $f a \rightarrow a \rightarrow Fix f$

	catamorphism	ana morphism	
generalized (f w \rightarrow w f) \rightarrow (f (w a) \rightarrow β)	$fa \rightarrow a$ $a \rightarrow fa$		g eneralized
	prepromorphism*	postpromorphism*	$(m f \rightarrow f m) \rightarrow (\alpha \rightarrow f (m \beta))$
	after applying a NatTrans	before applying a NatTrans	
	$(f a \rightarrow a) \rightarrow (f \rightarrow f)$	$(f a \to a) \to (f \to f) \qquad (a \to f a) \to (f \to f)$	
	paramorphism*	apomorphism*	
	with primitive recursion	returning a branch or single level	
	$f(Fix f \times a) \rightarrow a$	$a \rightarrow f$ (Fix $f \lor a$)	
	zygomorphism*	g apo morphism	
	with a helper function		
	$(f b \rightarrow b) \rightarrow (f (b \times a) \rightarrow a)$	$(b \rightarrow f b) \rightarrow (a \rightarrow f (b \lor a))$	
g histo morphism $(f h \rightarrow h f) \rightarrow (f (w a) \rightarrow a)$	histo morphism	futumorphism	g futu morphism
	with prev. answers it has given	multiple levels at a time	$(h f \rightarrow f h) \rightarrow (a \rightarrow f (m a))$
, ((3, 3,	$f(w a) \rightarrow a$	a → f (m a)	, (35, (35,))

refolds (build up then tear down a structure) algebra $g \ b \rightarrow (f \rightarrow g) \rightarrow coalgebra \ f \ a \rightarrow a \rightarrow b$

others synchromorphism ??? exomorphism ??? mutumorphism ???

hylo morp		rphism		
	cata;	ana		P. I
dynamorphism histo; ana		codynamorphism cata; futu		g eneralized apply the generalizations for both the relevant fold and unfold
	chronom histo	•		the relevant lold and uniold
Flanck a		iulu	-1	

Elgot algebra coElgot algebra ... may short-circuit while building cata; $a \rightarrow b \lor f$ a $a \times g b \rightarrow b$; ana reunfolds (tear down then build up a structure)

coalgebra g b → (a → b) → algebra f a → Fix f → Fix g

metamorphism

denerali

metamorphism generalized
ana; cata apply ... both ... [un]fold

combinations (combine two structures)

algebra $f a \rightarrow Fix f \rightarrow Fix f \rightarrow a$ **zippa**morphism $f a \rightarrow a$ **merga**morphism
... which may fail to combine

 $(f (Fix f) \times f (Fix f)) \vee f a \rightarrow a$

These can be combined in various ways. For example, a "zygohistomorphic prepromorphism" combines the zygo, histo, and prepro aspects into a signature like $(f \ b \to b) \to (f \to f) \to (f \ (w \ (b \times a)) \to a) \to Fix \ f \to a$

Stolen from Edward Kmett's http://comonad.com/reader/2009/recursion-schemes/

* This gives rise to a family of related recursion schemes, modeled in recursion-schemes with distributive law combinators

