A note about HW3's Lamp language.

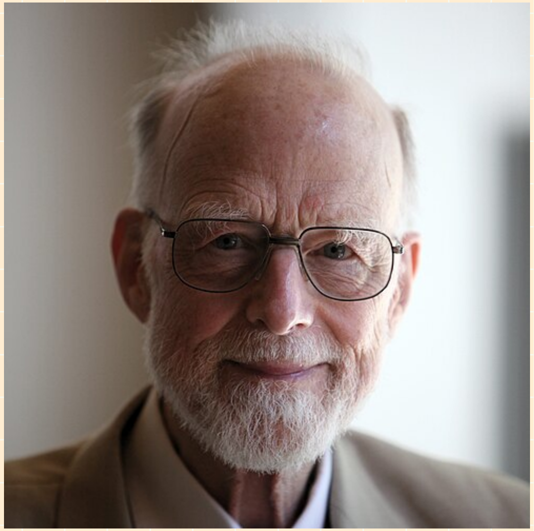
- We generalize pairs to n-ary tuples

Thought experiment:

- What if a library has a bug? Who's affected?
- What if a language has a "bug"? Who's affected?

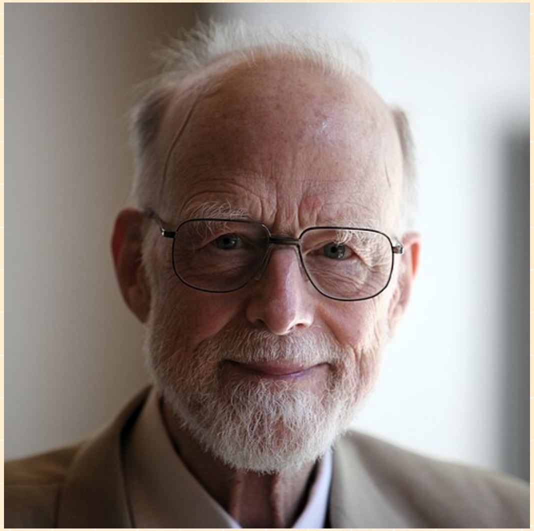
Today,

- The "billion dolloar mistake" in PL design
- Affects almost every mainstream language today
- One simple trick to fix it (sum types)


Tony Hoare (1934-)

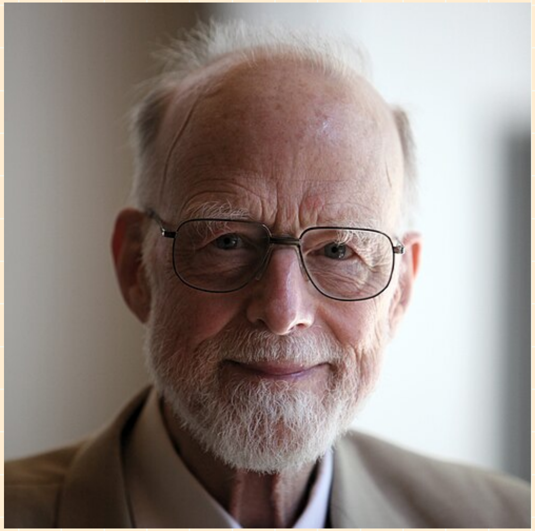
Inventor of:

1. Quicksort
2. Block structures
   - if { ... } while {...} (instead of GOTO)
3. Hoare Logic
   - huge influence in program verification
4. Null



Tony Hoare (1934-)

- I call it my billion-dollar mistake. It was the invention of the null reference in 1965.
- At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler.
- But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.
- This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.


Tony Hoare (1934-)

NULL appeared in language ALGOL X/ALGOL 60.

"Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors."

Languages influenced by the design of ALGOL:

- C, C++, Java, JavaScript, Python, Go

Languages with null references/pointers

Language	Notes
Java	`null` is valid for any reference type (not primitives).
JavaScript	`null` can be assigned to any variable, any object field.
C	`NULL` can be assigned to any pointer type.
C++	`nullptr` / `NULL` can be assigned to any pointer type.
C# (pre-nullable types)	`null` is valid for all reference types (value types need `Nullable<T>`).
PHP	`null` can be assigned to any variable.
Ruby	`nil` is an object and can be assigned anywhere.
Python	`None` can be assigned to any variable.
Perl	`undef` is the uninitialized value, works in all scalar contexts.
Objective-C	`nil` is a valid value for any object pointer.
Lua	`nil` is the default value for any missing variable or table entry.
Visual Basic	`Nothing` can be assigned to all reference types.

What's the problem with null?

```
T* object = new Object( … );

 …

if (object ≠ nullptr) {
    …
    do something about object
} else {
    …
    do something about object
}
```

Compiler only knows:

1. AST
2. Types of expressions

and nothing else.

We're going to fix null by generalizing booleans in 2 steps.

boolean values

true, false

boolean type

Bool

introduction form (how to make a boolean?)

elimination form (how to use a boolean?)

if

Generalization step 1: allowing more than just true/false

RGB

enum values

$\underbrace{\text{'red}}, \underbrace{\text{'green}}, \underbrace{\text{'blue}}$

label / tag

enum type

+{'red, 'green, 'blue}

introduction form (how to make an enum?)

elimination form (how to use an enum?)

```
switch (e) {
   case RED :
      - - -
```

Examples of phenomena that can be modelled by enum type

- Color channels
- Boolean
- Days of week
- Months
- Year in school
- Any finite set

```
+{'true, 'false}
+{'mon, 'tue, 'wed, ...}
+{'Jan, ...}
+{'freshman, 'sophomore, ...}
```

```
switch (e) :
   case 'true : ...
   case 'false ...
```

Generalization step 2: every enum value contains one piece of data

We call this "sum type" = enum on steroids

$\underbrace{\text{sum values}}$

'a(1)   'b(False)

sum type

+{ 'a : Nat,
'b : Bool }

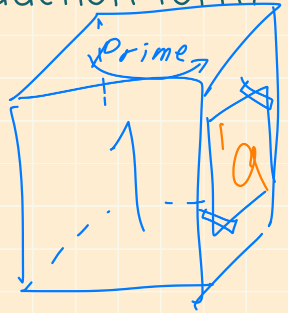
introduction form (how to make a sum value?)

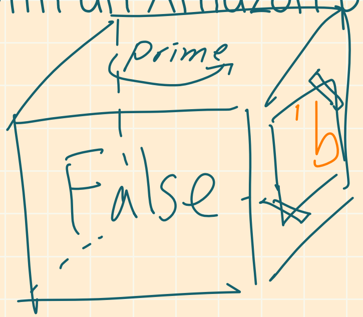
elimination form (how to use a sum value?)

Mental model of sum types:

'a (1)   'b (False)

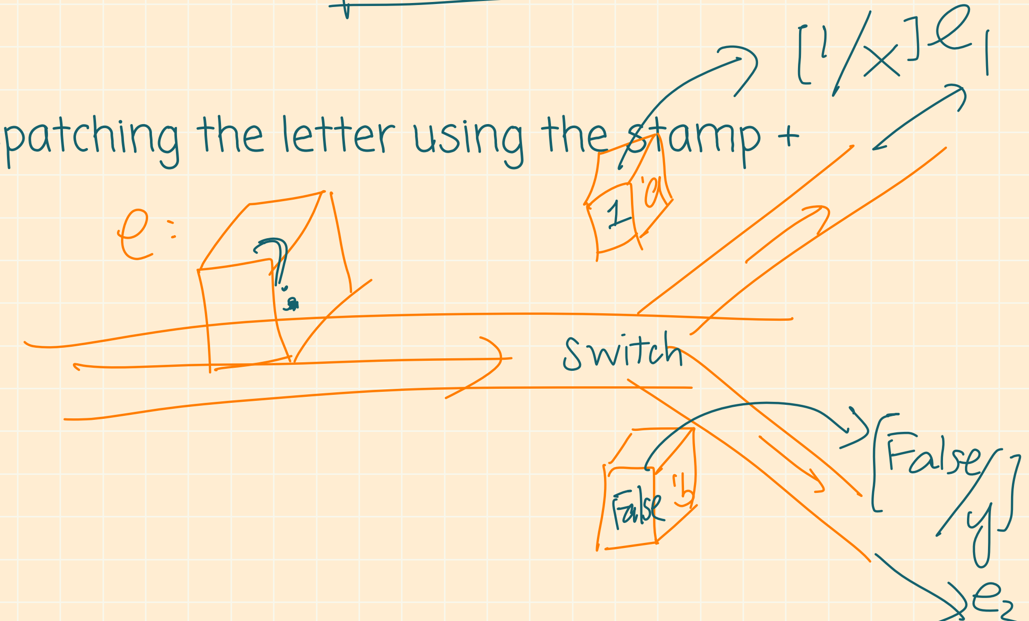
- introduction form = wrapping data with an Amazon package





- elimination form = dispatching the letter using the stamp + unwrapping

```
switch (e) {
   'a x :   e1 ,
   'b y :   e2
}
```



(Live coding) Examples of sum types:

- Define a color type. Color can be (RGB), value represented by a Nat.

- Define an unsafe division function. If divisor = 0, return 0.
- Wrap unsafe division into a safe division function.

- Using the safe division, define a safe quotient/modulo function.

Previously, types = over-approximation to rule out undefined behaviors

Now, types = logical specification of your data/program

Powerful type system = compiler can rule out logical bugs for you

Examples:

(Nat, Bool, Nat → Nat)

Product type (t1, t2, ..., tn) = type of tuple values

- What's the logical essence of product type?

AND = $\wedge$

T

Enum type +{'success : Bool, 'divByZero: Nat} = type of enum labels

- What's the logical essence of enum type?

'success True

OR

'divByZero 0

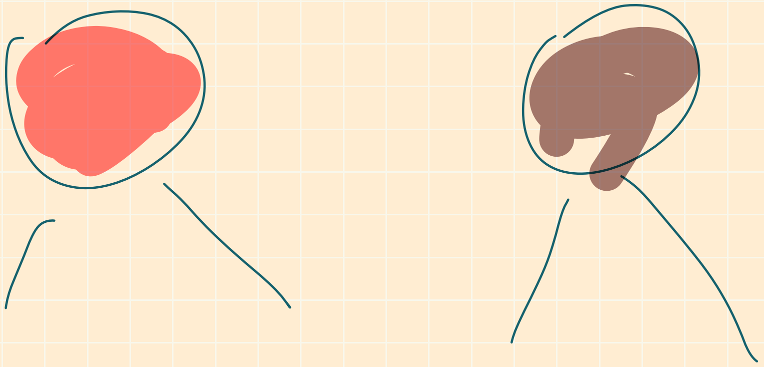
(also : +{'success: Bool})

Let's use AND and OR to model common data structures

non-recursive

- A pair of (Nat, Bool) is = Nat AND Bool
- A boolean is = True OR False
- A color channel is = R OR G OR B
- A month is = . . .
- A datetime is = $\underbrace{\text{Nat}}_{\text{hour}}$ AND $\underbrace{\text{Nat}}_{\text{min}}$ AND $\underbrace{\text{Nat}}_{\text{sec}}$
- A division returns = Nat OR Nat
- A singly linked list is = Empty OR $\underline{\text{(Nat AND linkedlist)}}$
- A binary ^{n} tree is: = Empty OR Nat AND BinaryTree AND BinaryTree
- A ternary tree is:
- A n-ary tree (b-tree) is:
- A red-black tree is:

recursive



Empty OR $\overset{\text{(red)}}{\text{Nat AND RBT}}$

AND RBT

OR $\overset{\text{(black)}}{\text{Nat AND}}$

RBT AND RBT

What is a Lamp AST?

AST is $e ::= n \overset{\text{OR}}{|} b$

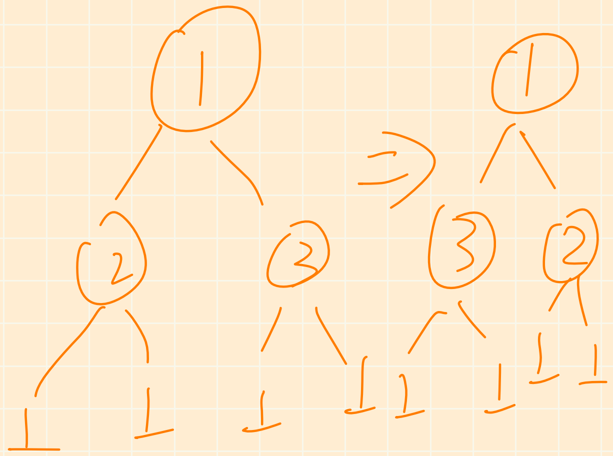
$\underset{\text{OR}}{|}$ $e_1 \underset{\text{AND}}{+} e_2$ $\underset{\text{OR}}{|}$ if e_1 $\underset{\text{AND}}{\text{then}}$ e_2 $\underset{\text{AND}}{\text{else}}$ e_3

Recursive types: type equations of the form $X = t$, where

- X is an abbreviation
- t is a type where X can appear

Live coding: defining recursive types in Lamp

- Singly linked lists
- Binary search trees



Pattern matching:

Combining the elimination forms of products and sums.

```
let (x, y, ... ) = e1 in
e2
```

```
switch e0 {
  'l1 x: e1,
  'l2 x: e2,
    ...
}
```

A pattern match is a list of branches.

- Branch = pat : expr.
- Each pat describes the expected "shape" of the data
- If the actual data matches the expected "shape", the expr is executed
- Go through branches sequentially to find & execute the first match

```
match e0 {
  pat1: e1,
  pat2: e2,
  ...
}
```

Pattern matching: Formal definition

```
expr   ::=  ...  | match expr { (branch)* }
branch ::= pat: expr
pat ::=
        | (pat1, pat2,  ... )
        | 'label pat
        | x
```

Operational semantics (1st attempt)

Example: boolean negation

Operational semantics (2nd attempt)

```
σ ::= (empty)
    | σ, x:v
```

```
value ⋈ pat → σ
```

Means **`value`** matches **`pat`** by using dictionary **σ** to map pattern variables to values