CS131 Week 3

Week 3 Lecture 1

==========

Language implementation
- where is code running
- how is it developed
- batch environments (premium on no typos)
- interactive (read-eval-print loop)
- embedded apps
- real-time
- distributed/web/apps

Functional Programming
OCaml

Compiler vs Interpreter

Compiler	**Interpreter**
x = x+5; -> compiler	keeps source code in RAM
	executes it directly
	more portable (if written in portable language)
	simpler
just translates p = &a[i] into leaq(%rax, %rbx, 4), %rdx... is there out of bounds here?	better error checking (out of range p = &a[i] ??)

Compromise between Compiler and Interpreter
Insns for an abstract machine for language

Keep stripped down version
- **bytecodes** (8 bits each code: cheap)

Deletes all the comments and unnecessary stuff

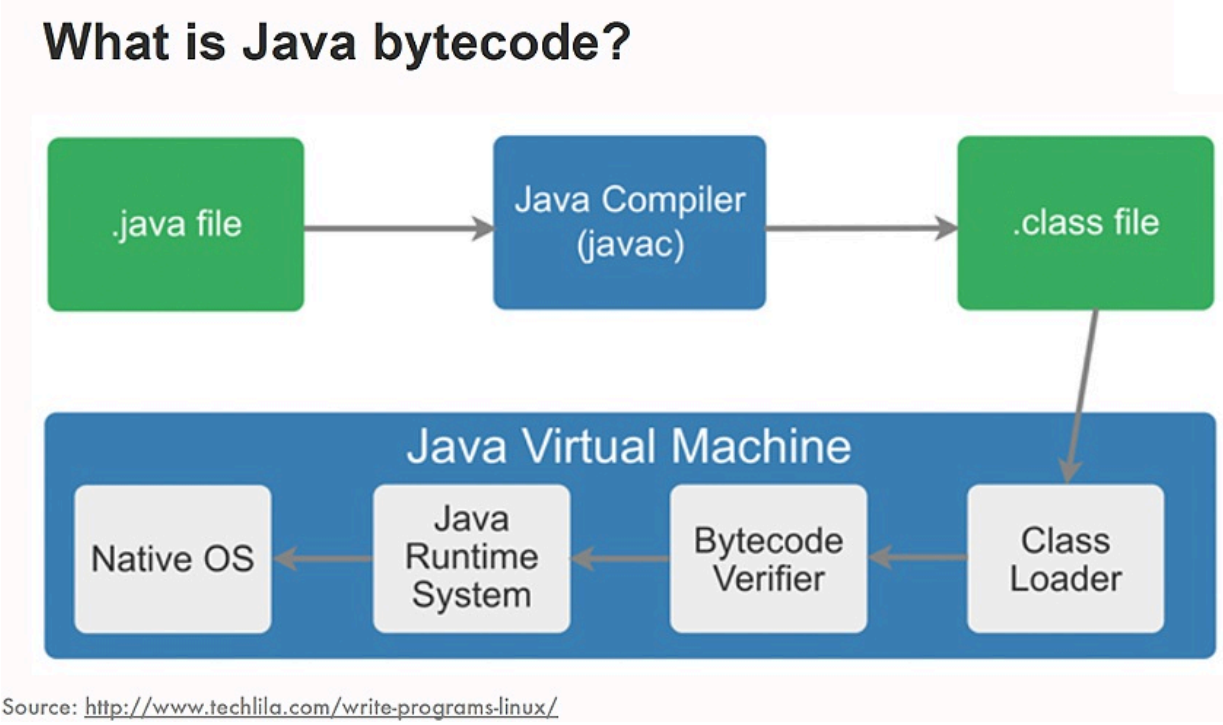
Foo.java -> Foo.class -> java executable bytecodes
^ class Foo {} ^ bytecodes ^ java bytecodes

Problem with early bytecodes: LOSING BENCHMARKS
FIX: **Just In Time** compilation: moving toward compiler side of spectrum

Foo.java -> Foo.class -> java executable bytecodes -> JUST IN TIME COMPILER (machine code)

IDEA: Use interpreter while developing, but switch to compiler during production

- interpreters slow
- compilation is fast
- developers can waste time with interpreters, but users want quick compilation and use



Source: http://www.techlila.com/write-programs-linux/

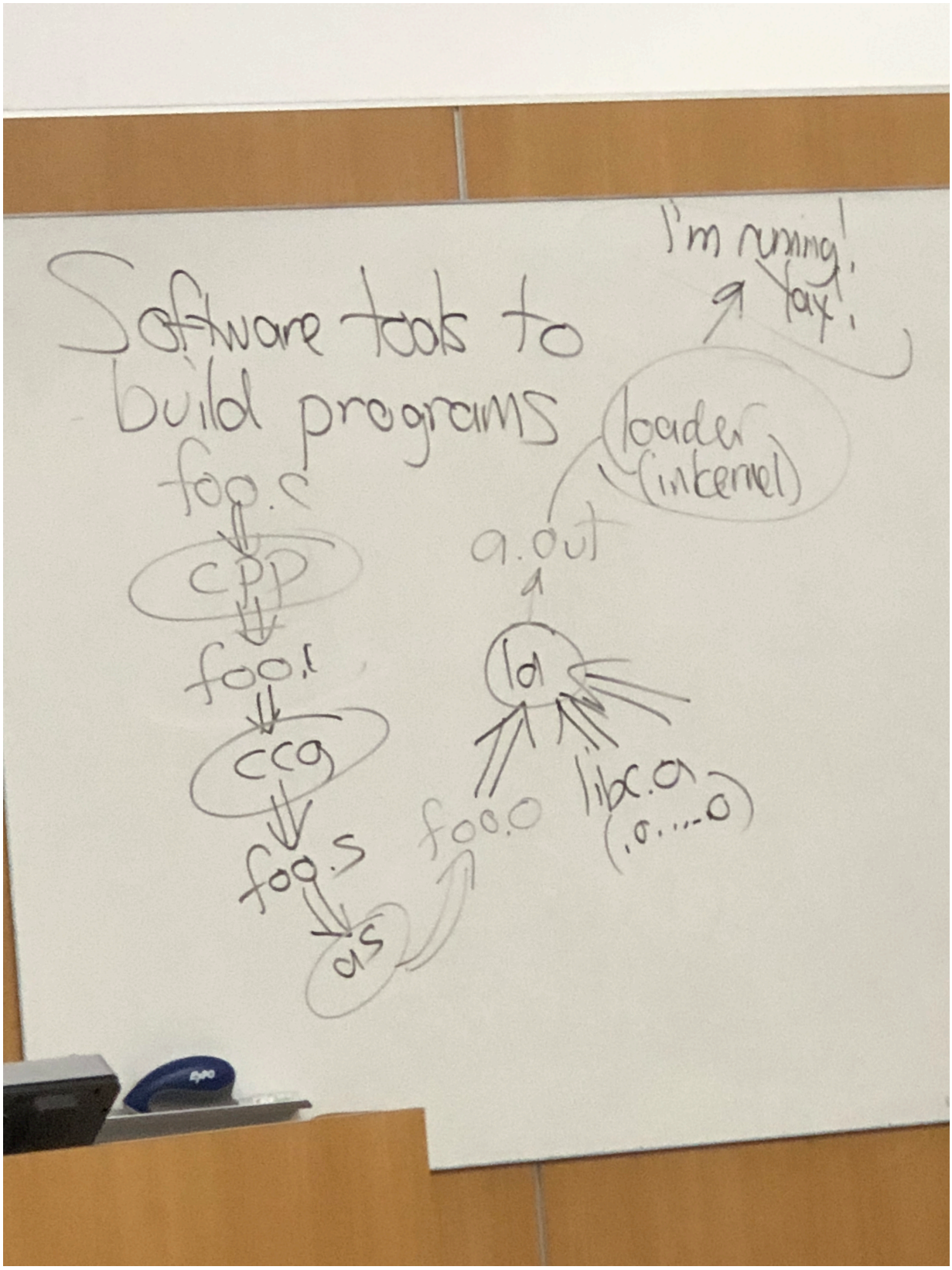
*Java Runtime System ~= JIT ???

JIT also has an advantage that it can see exactly what you're running, and then can choose to compile things well. gcj (Java version of gcc) doesn't know what's happening so it compiles and may be buggy.

- If JIT is buggy, u fucked

JIT uses # of execution counts, less accurate but cheaper to measure than time of execution

Software Tools Approach to Building Programs



Pros:

- modularity
- replacability of components

Cons:
- long ass chain of modules
- hard on developers

IDE (Integrated Development Environment) - Competing Approach to Building Programs
Smalltalk
- you click a button and it compiles and runs for you

IDEs can be tightly integrated or loosely integrated, like the modular approach above e.g.
- Smalltalk is tightly integrated
- Eclipse is loosely integrated

Dynamic linking is "bridging the gap": You can link some things together at runtime rather than before compilation: has benefits of JIT seeing how things play out before deciding how to link/what to link

OCaml (ML variant)
- static type checking (checks data types at compile time)
  - want this to make code more reliable
  - like C++, Java
  - unlike Python, Javascript, etc..
- you don't need to write the types down usually (surprising, since usually #1 implies no #2)
- type inferencing scheme
- has garbage collector (garbage collector)
- good support for higher-order functions

```
val x : int = 1369
    ^    ^        ^
 name type   value
```

```
# if 0 < 1 then "a" else 29 ;;              (* ERROR, returns 2 types *)
# if 0 < 1 then ("a", 0) else ("", 29)      (* correct! *)
```

```
# 1, "a", 5.7
-: int x string * float = ...
```

Lists vs Tuples

Lists	**Tuples**
homogeneous	heterogeneous
any length	length fixed

Patterns (how to match certain types)
constants:
- 0
- “a”

variables: (match anything)
- x
- y
- _

tuple
- P, Q

nonempty list
- P::Q
- P::(Q::P)

empty list
- []

int * int

```
match E with
| 0, x -> x
| y, 1  -> y
| x     -> x
```

— 1/23/18

Week 3 Lecture 1

==========

Using static type checking to help learn OCaml
- Ocaml will tell you what type your function/assignment returns
- will also tell if you have a function or variable

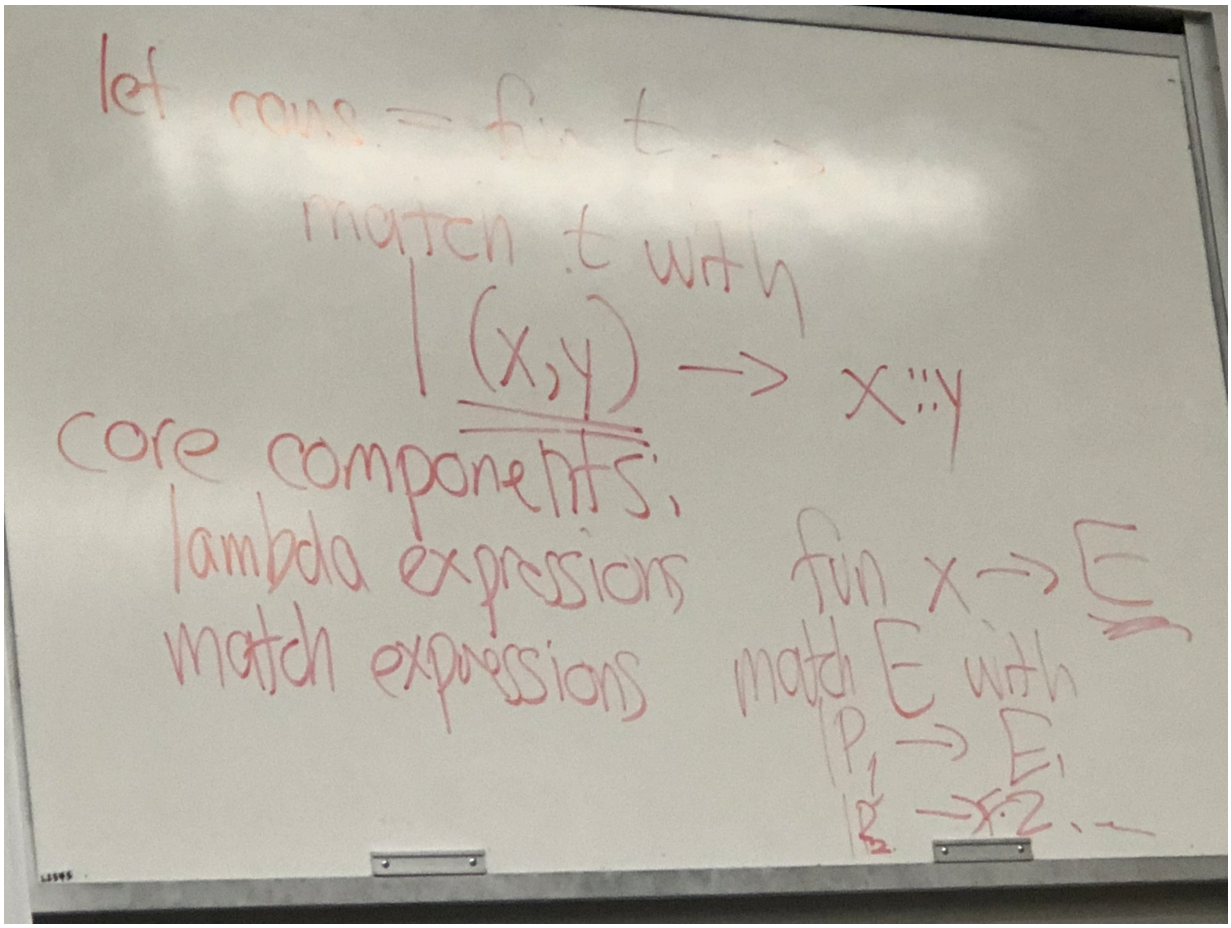
```
let car(x::_ ) = x;
-: var car: ‘a list -> ‘a = <fun>
let car = fun x -> match x with | (x::_) -> x
# car [] ;;                (* declaration *)
```

currying
- taking a function with multiple arguments and turning it into higher order function
- your declared function creates a function that calls another function that takes in another variable and so on and so on

NOTE: he's calling "curried cons" "ccons"

let ccons x y = x::y ;;
-: var ccons : 'a -> 'a list -> 'a list = <fun>
let ccons1 = ccons 1 ;;
-: var ccons1 : int list -> int list = <fun>

*couldnt type fast enough, so this is pic of last board



Eggert's code for intlist curried cons example

```
// name type
typedef struct intlist {
        int n;
        struct intlist *next;
} intlist

static int isave;      // NOTE: bug for later

// NOTE: "filil" = function int list -> int list
```

```
filil ccons(int i) {
        ...
        return ocons; // ordinary cons
}

// ordinary cons called above quickly defined here
intlist ocons(intlist x) {
        intlist p = malloc();
        p -> i = i;
        p->next = (x)
        return p;
}

filil f = ccons(39);
intlist l = f(NULL);
intlist ll = f(l);
```

Eggert talks about bugs in the program:

The static variable is not thread-safe, and also won't work in single threaded variable.

- the issue is to store a variable for arguments that are currently passed in a safe place that can be accessed as long as variable needing it is in play
- problem is, cannot store as static because every call to the function will overwrite this argument stored in static variable

Identifying Bugs in your code using static type checking and fixing them

```
let rec reverse l =
        match l with
        [] -> []
        | h::t -> (reverse() @ h ;;
-: var reverse : 'a list list -> 'a list = <fun>
(* problem: 'a list list should just be 'a list *)
```

- when compiler sees @ h, it thinks we're concatenating list with list (h is list), so when u see h::t you know h is an element of list l, so compiler deduces l is list of lists
- Didnt even need to run program to debug

FIX ?

```
let rec reverse l =
        match l with
        [] -> []
```

```
        | h::t -> (reverse() @ [h] ;;
(* slow as fuck *)
```
- hella slow, so need to do more fixes to fix speed bug

FIX ?

Solve a "harder" more general problem in faster runtime

```
let rec revapp l a =
        match l with
        [] -> a
        | h::t -> revapp t (h::a) ;;
let reverse l = revapp l [] ;;
```

*can also say

```
let rec revapp a = function
        [] -> a
        | h::t -> revapp (h::a) t ;;
let reverse = revapp [] ;;
```

```
(* "syntactic sugar" *)
function | P1 -> E1 | P2->E2 | ...
==> fun x -> match x with | P1 -> E1 | P2 -> E2
```

'fun' is for currying

'function is for pattern matching

```
i.e. fun x y -> x + y + 1
=> fun x -> ( fun y -> x + y + 1 )            (* currying *)

i.e. function | [] -> 0 | _::_ -> 1
=> fun x -> match x with | [] -> 0 | _::_ -> 1          (* pattern matching *)
```

*most of the time, you can use fun or function and it doesn't really matter

```
i.e. fun x -> x + 1
function x -> x + 1
fun y -> match y with | x -> 1
```

*ONLY USE FUNCTION IF MATCHING SOME NONTRIVIAL PATTERN

```
let rec minlist = function
        [] -> 99999999999999999 (* big ass number so most # < this *)
        | h::t -> let m = minlist t
```

```
in
        if h < m then h else m              (* note: this is hardcoded. could have a more
general version of function, minlista (below) *)
-: val minlist : int list -> int = <fun>

let minlista inf lt = function
        [] -> inf                                   (* inf is type 'a *)
        | h::t -> let m = minlista inf lt t
in if (h lt m) then else m
-: val minlista : 'a -> ('a -> 'a -> boolean) -> 'a list -> 'a
```

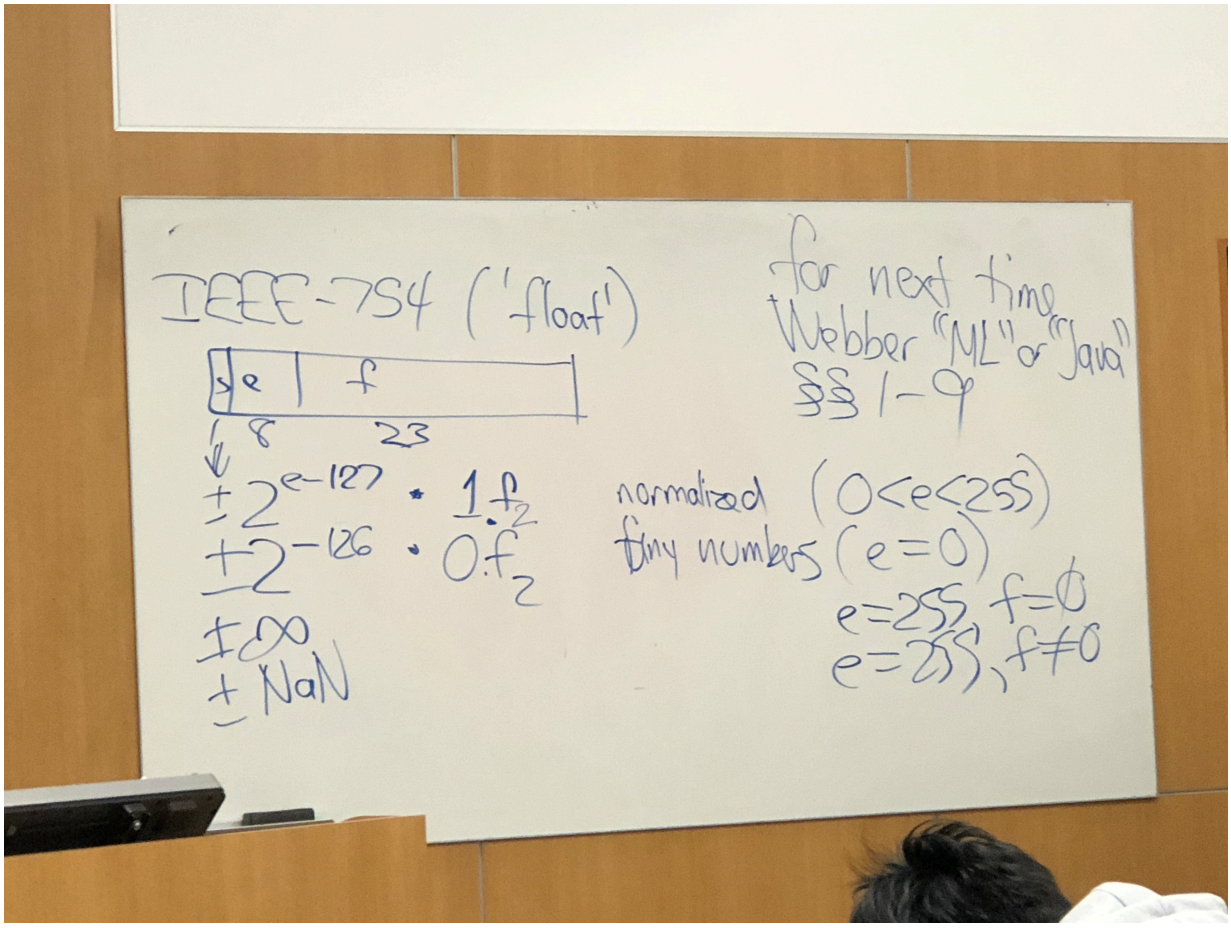
Types

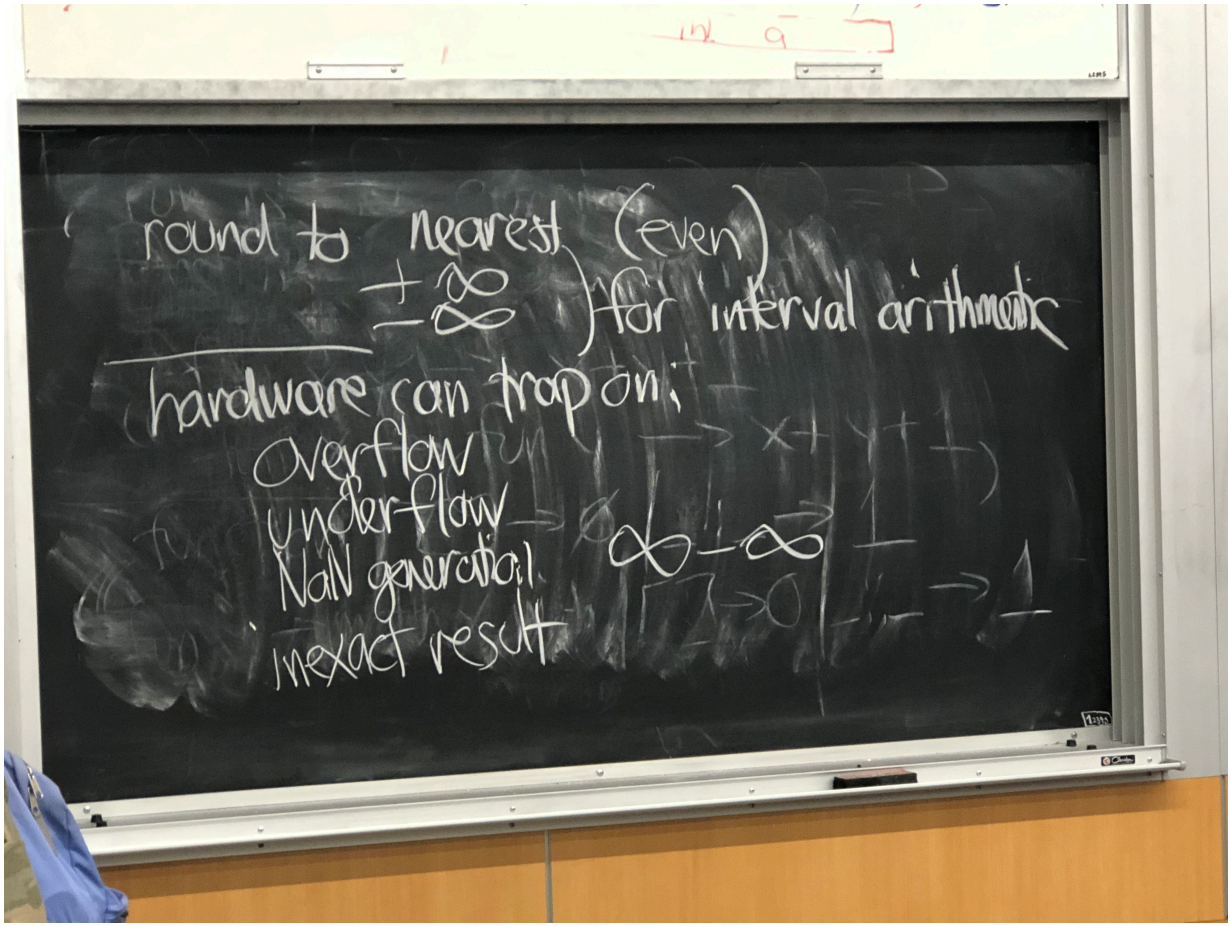
What is a type?

1. a set of possible values
    1. Therefore, subtype = subset
2. a set of operations on these values
    1. defines what you can do with these values

Primitive vs constructed types

Primitive	Constructed
builtin types	defined by users (e.g. arrays)
int, float, long, boolean, char	newer, flakier ??
portability issues: (int width -2^31 ... 2^31 - 1) (char signedness? 0...255 unsigned char, -128...127 signed char) (floating point)	


IEEE-754 ('float')
s
e
f
8
23
$\pm 2^{e-127} \cdot 1.f_2$
$\pm 2^{-126} \cdot 0.f_2$
$\pm\infty$
$\pm$ NaN
for next time
Webber "ML" or "Java"
§§ 1-9
normalized (0<e<255)
tiny numbers (e=0)
e=255, f=0
e=255, f≠0


round to nearest (even)
±∞
−∞
for interval arithmetic
hardware can trap on:
overflow
underflow
NaN generation
inexact result

— 1/25/18