Course Logistics

1.	4 Quizzes (no makeup Qs)	12%
2.	4 Assignments (mandatory - all of them)	28%
3.	Minor Exam	30%
4.	Major Exam	30%

- Attendance in Exams Cumpulsory

Introduction to Logic & Functional Programming

- Quickly about the course
  - Various programming paradigms

    functional, imperative, logic, & OOP

    . for types

    . technique for language description

Model of Computation

- Expression Interpreter i.e. function applied to subexpressions
- Sequence of actions on an underlying m/c
- Constraint calculus on relations ↳ deduce new inferences
- independent objects to interact by sending msgs to each other

  - Language Description & Implementation
    - syntax + definitional interpreter

      ↳ lambda calculus] → A convenient vehicle to study hofunctions, types etc.

  - Concepts from compilers and Software Engineeri

o Language can be described by writing a def^n interpreter Eg: LISP

Poll

- How many know functional programming
- How many know logic & logic programming?
- Have you used recursion?
- How comfortable are you with mathematical induction?
-

Lec 1

And what is computable?

- What is a computation?

  → A: a finite sequence of transformations by means of predefined rules on finite & discrete data

  → specified with the help of programs

eg: $f(x) = x+1$,

Q: What is a non-computation?

→ writing a poem?

- What is an algorithm?

  → a finite seq. of instr. solving a problem

- What is a program?

  → Representation of an alg. in a lang.

And programming?

→ Logical aspects of program organization

→ D.S. choice, Modularity, Reuse etc.

The choice of programming language is important!

→ It determines the kind of computations that can be carried out!

Eg: Computational model consists of a ruler + compass

→ what we can compute?

→ Bisect an angle, $\sqrt{n}$, ...etc.

→ what we cannot do?

→ trisect an angle

Need additional things in our model of computation such as protector.

Other computational tools

- Abacus (still used in Japan)
- Paper & pencil
- Stick & stones
- ⋮

→ Programming languages (model of computation) that can be turing complete with goto and if-then-else

└→ Computationally universal → anything that is computable via an algorithm can be made to run on a turing m/c.

└→ HTML turing complete?

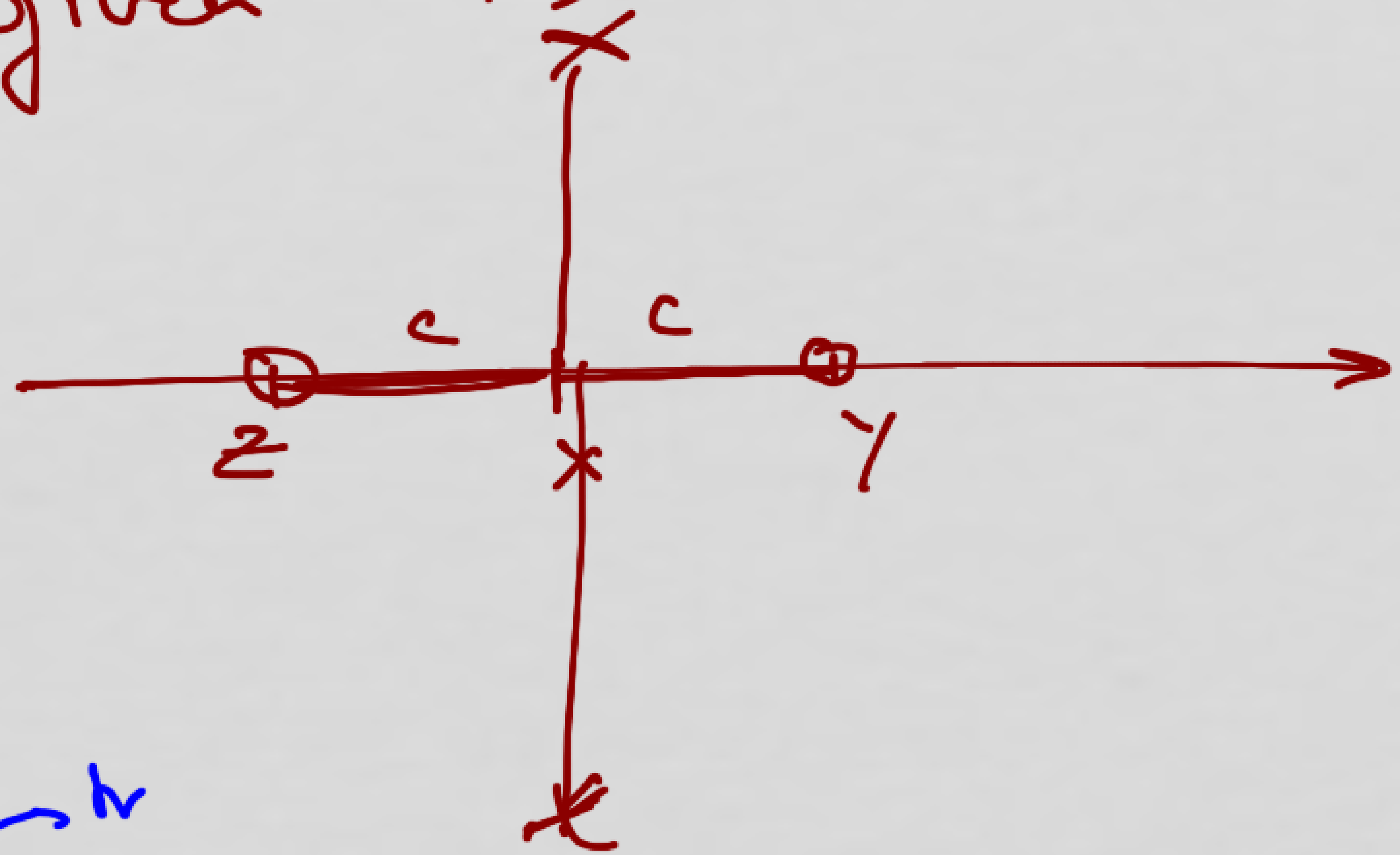
└→ regular expression turing complete?

In summary every computational model must have

- primitive ops & exprs which represent the simplest objects with which the model is concerned
- Methods of ~~ab~~ combination: how to combine primitive ops to get compound exprs

- Methods of abstraction: how to use compound exprs as named units.

Eg: Perpendicular at a pt (x) given a line



Steps

1. Mark Y & Z at c units from X
2. Draw circles $C_1(Z, 2c)$ $C_2(Y, 2c)$
3. Join the pts of intersectn of two circles

Using ⊥ constr to construct a square → Method of abstraction

Program specification

- How do we specify what to expect from a program?
- How do we map what we expect from the program & what the program really computes?
- How do wer ensure program is correct vis-a-vis its specification

Ans: Rigorously answered by means of formal mathematical specification & a formal relationship betn spec & prog.

PL History

- → invented to make easier use of m/c
- → can be both higher-level & general purpose

I Machine Language → lowest level (Von Neumann m/c 1946)
→ virtually unintelligible

II Assembly Language → variant of Machine language but with Mnemonics, [i.e. names of m/c ops symbols for values, storage locations etc.

Readable but tied to a m/c arch. [No m/c independence]

III Fortran (formula translator)

↓

first compiler, [therefore m/c independence]

↳ portability

Availability of libs

IV LISP (list processor)

(interpreters)

Exp. evaluators

i.e. symbolic manipulator

Interpreters

↳ High level m/cs

Prolog

(Abstract m/cs)

Logical inference

V Algol

Compiler on a V.M. ← PASCAL

C/C++

Compilers are translators

Functional Programming

- S/w becoming complex, becoming more important to structure it well

- Conventional languages place conceptual limits on the way the problems can be modularised

- In F.P. – functions are treated as mathematical functions

  which

  means

- No assignment statements

$x := x + 1$

Referentially transparent

→ expr ⇔ values

No change in the behavior of the program

└→ thus a variable given a value can never change

└→ variables are treated as constants

** Major source of bugs → side-effects

└→ More generally, no side-effects

└→ As a result common compiler opt. such as subexpression elimination, code movement etc do not require fix pt data-flow analysis

Also → No sideeffects ⇒ no need to ~~define~~ prescribe flow of control

→ programs become tractable mathematically

- No iteration loops

  → instead recursion is used

  [theory behind it is induction]

- Higher order functions

  → leads to compact & concise code

Note: I/O is inherently imperative