From Turing machines and Lambda calculus to today's programming languages

- Turing machines can be seen as the natural ancestors of procedural and object oriented languages
- lambda calculus formalizes the key operations in functional languages (although function notation as used in mathematics has been around for much longer)

Turing machines $\Rightarrow$ Random access machines $\Rightarrow$ today's computers

- The Turing machine's tape and store is replaced by directly addressable memory
- machine and the assembler languages follow
- procedural languages (compiled or interpreted)
- object oriented languages help encapsulating state and reuse code via inheritance mechanisms

Combinators and Lambda Calculus

- functions, via application and composition
- substitution mechanisms instead of overwriting of memory state
- simpler equivalent formalism: combinators
  - S f g x = (f x) (g x)
  - K x y = x
  - I x = x
- naming of a function separated from specification of how a function operates: λ x . x

Turing Machines, Lambda Calculus, Combinators all have the same computational power

- they can *emulate* each other: an interpreter for any of them can be written in another formalism
- they are all called *Turing Complete* formalisms
- the same applies to *all* programming languages
- what distinguishes them is *expressiveness*

Links to explore

- Turing Machines Turing Machines - more
- RAM and RASP machines
- Lambda Calculus Lambda Calculus -more
- SKI combinator calculus
- Turing Completeness