Recursion equally powerful to iteration, and as efficient in cases where you can use tail recursion.

Mechanical transformations back and forth often more intuitive (sometimes less)
naive implementation less efficient no special syntax required
fundamental to functional languages like Scheme
tail recursion (*OCaml:*)
```ocaml
let rec gcd b c =
  if b = c then b
  else if b < c then gcd b (c - b)
  else gcd (b - c) c;;
```
implemented as
gcd (b c)
```
start: if b = c return b
  if b < c c := c - b goto start
  if b > c b := b - c goto start
```
changes to create tail recursion (e.g. pass along an accumulator)

(*OCaml:*)
```ocaml
let rec summation f low high =
  if low == high then f low
  else f low + summation f (low+1) high;;
```
becomes
```ocaml
let rec summation2 f low high st =
  if low == high then st + f low
  else summation2 f (low+1) high (st + f low);;
```
and then
```ocaml
let summation3 f low high =
  let rec helper low st =
    let new_st = st + f low in
    if low == high then new_st
    else helper (low+1) new_st
  in
  helper low 0;;
```
More generally (absent an associative operator), pass along a continuation.

This is perfectly natural to someone used to programming in a functional language. Note that the summation example depends for correctness on the associativity of addition. To sum the elements in the same order we could have counted down from high instead of up from low, but that makes a more drastic change to the structure of the recursive calls.

There is no perfectly general algorithm to discover tail-recursive versions of functions, but compilers for functional languages recognize all sorts of common cases.

Sisal and pH have "iterative" syntax for tail recursion:
```sasal
function sum (f : function (n : integer returns integer),
  low : integer, high : integer returns integer)
for initial
  st := f (low);
while low <= high
  low := old low + 1
  st := old st + f (low)
returns value of st
end for
end function
```
The Sisal compiler was really good at finding tail recursive forms.

Concurrency specifies that statements are to occur (at least logically) concurrently concurrency is fundamental to probably half the research in computer science today
Subject of chapter 13

Nondeterminacy choice "doesn't matter" periodically popular, promoted by Dijkstra for use with selection (guarded command syntax) can apply to execution order as well useful for certain kinds of concurrency

Process server do
  receive read request ->
  reply with data
  []
  receive write request ->
  update data and reply
od
also nice for certain axiomatic proof schemes raises issues of "randomness", "fairness", "liveness", etc.