Declarative Programming

Differences to Imperative Languages

- focus is on what to do, rather than how to do it
- higher level of abstraction
- easier to use powerful programming techniques
- clean semantics: can do things with declarative programs you can't do with imperative ones

Paradigms

- **Imperative:** based on **commands**, as instructions and statements
  - commands are executed
  - commands have an **effect**: update computation state. Later code may depend on this update
- **Logic:** based on finding values that satisfy a set of **constraints**
  - constraints may have 0 or many solutions
  - constraints have no effect
- **Functional:** based on evaluation of **expressions**
  - expressions are evaluated
  - expressions have no effect

Side Effects

- code has a **side effect** if, in addition to producing a value, it also modifies some state, or has an observable interaction with calling functions/outside world.
- examples
  - modify global/static variable
  - modify an argument
  - raise an exception
  - write data
  - read data
  - call other functions that have side effects

Destructive update

- imperative languages: natural way to insert an entry in a table is to modify the table in place
    - destroys the old table
- declarative languages: instead create a new version of the table, while the old version remains
    - drawback: language has to work harder to recover memory and ensure efficiency
    - benefit: don't need to worry about what other code is affected by the change
        - can keep previous version for comparison/undo
        - **immutability** makes parallel programming significantly easier