High Level Programming

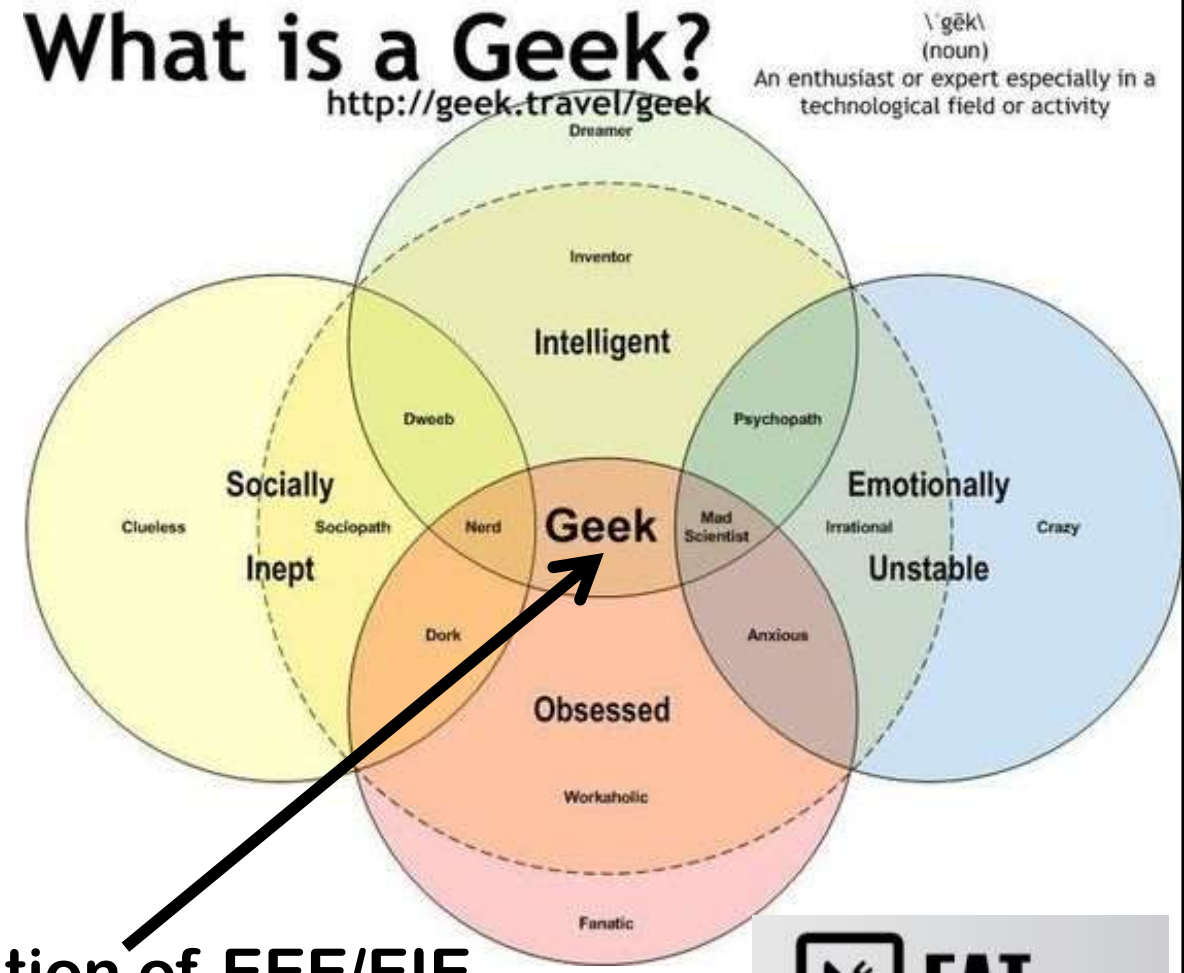
Tom Clarke: Spring 2017
EE3 & EIE3

Introduction

Lecture 1

Prerequisites

- **Interest in programming**
  - You enjoy writing and analysing programs
- **Interest in concepts & maths** that underlies programming
  - You found recursion a really neat idea!
- If you have done significant **programming outside EEE/EIE classes** using Java/C#, etc:
  - You may have some object oriented (OO) "ways of thinking" to unlearn
  - This module will improve your ability program in all languages (including OO)
- If you have done **no programming outside EEE/EIE classes**
  - That is no problem for this module
  - EEE students should have been in the top 1/3 of the class for programming



Definition of EEE/EIE surviving years 1 & 2??

NOT a good idea!

Aims

- **Learn to use new modern programming language features**
  - Programs as functions
  - Types as interfaces
  - Rich type systems
- **Solve programming problems more efficiently**
  - Model problem domains with rich type system
  - Use function composition to break down larger problems
  - Property-based testing
- **Practice problem-solving in groups writing big programs**
  - Gain experience as part of a group
  - Define interfaces, modularise
  - Top-down specification of code
- **Have fun!**

Coursework (weeks 1-4)

- **Active Learning**
  - Learn through doing structured examples or project work
  - Language reference is self-learnt
  - Lectures provide introduction and motivation
- **Programming worksheets to learn F# language**
  - Combine examples, analysis and coding
  - Small tick-marked deliverables with tight deadline for each sheet provide structure

[From Communications of ACM 2015]
Be It Resolved: Teaching Statements Must Embrace Active Learning and Eschew Lecture

Last year, the *Proceedings of the National Academy of Science* published a meta-analysis of 225 studies. The conclusion appeared as the title of the paper, *Active learning increases student performance in science, engineering, and mathematics*.

Project work (weeks 6-11)

Timetable

- **Teams of 4**
- **Last 6 weeks of Term**
  - 8 hours / week
- **Default: Reimplement VisUAL – an ARM assembler and simulator**
  - Use FP techniques to write an emulator quickly and with few errors
  - Emphasise modularity and reusability
  - Use modern web technology
  - Scope for group-defined customisation
- **Assessment**
  - Group demo, long individual interview, submitted code
  - Clear individual responsibilities
- **Technology used in default project**
  - F# programming language
  - FSCHECK test framework
  - FABLE F# → Javascript transpiler
  - CodeMirror programmer's editor component
  - HTML for GUI (easy!)
- **Self-proposed projects allowed**

Challenge Marks

- Coursework assessment
  - Allow 10% of module marks for outstanding self-motivated non-formulaic work
  - Less than 1/3 of students will get these marks
  - Examples:
    - Outstanding project work
    - Something else done in module for interest not marks which is outstanding
  - Prevents automatic moderation
  - Allows creativity to be rewarded

Content

- Functional Programming (FP) and a little Object oriented Programming (OOP) in F# (50% of module)
- Garbage collection & lambda calculus implementation
- Testing using semi-automatic test software
- OOP vs FP trade-offs
- Client-side web programming via F# -> Javascript transpiler
- (F* and dependent-typed functional languages for program specification and verification)

OOP vs FP

OOP makes code understandable
by **encapsulating** moving parts

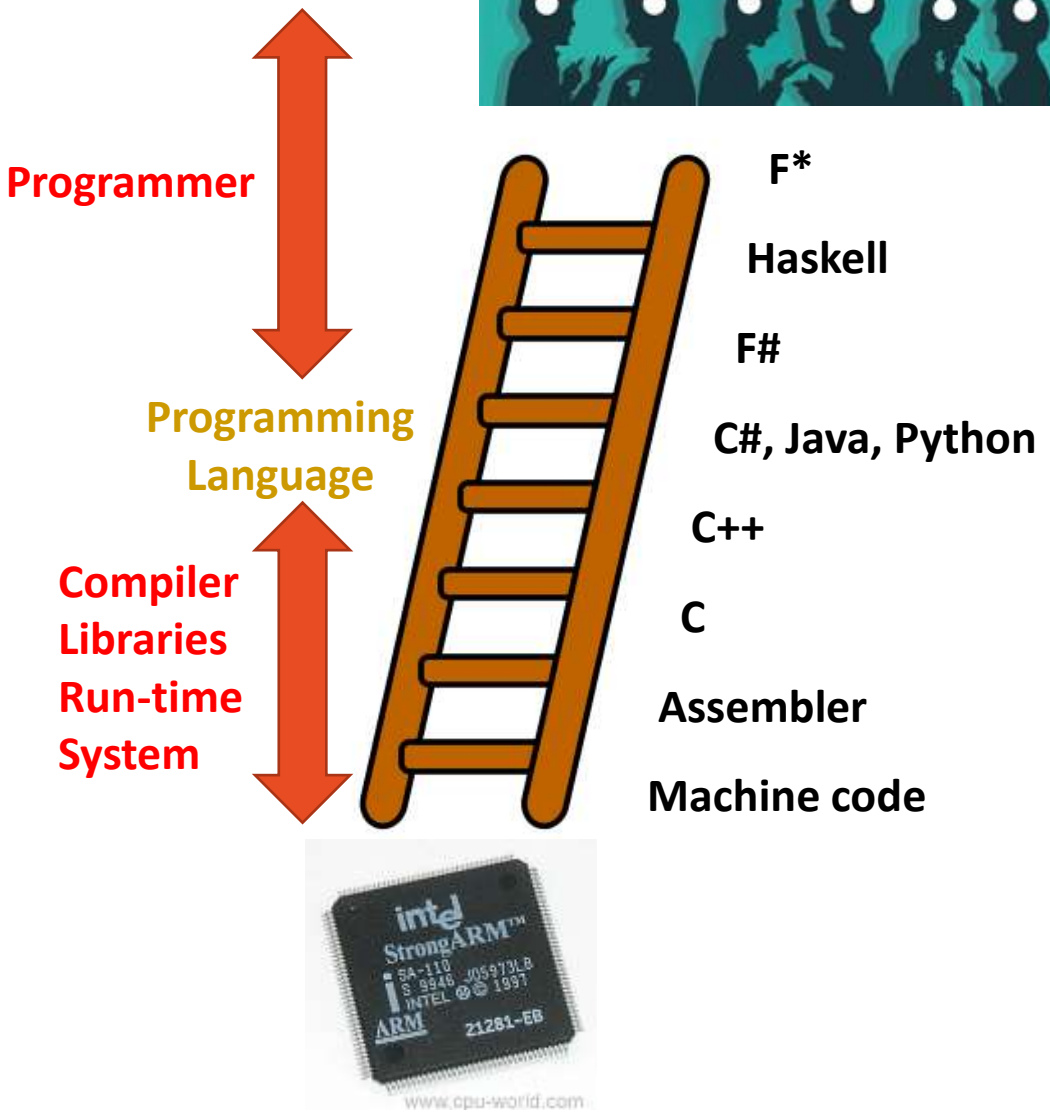
FP makes code understandable
by **minimizing** moving parts

- Michael Feathers

More

What is High Level Programming?

- Let programming language do more work
- Support **abstraction** to make large/complex programs tractable
- Provide **type system** to document and specify operation and prevent errors
- Implement built-in **data structures** managed by compiler



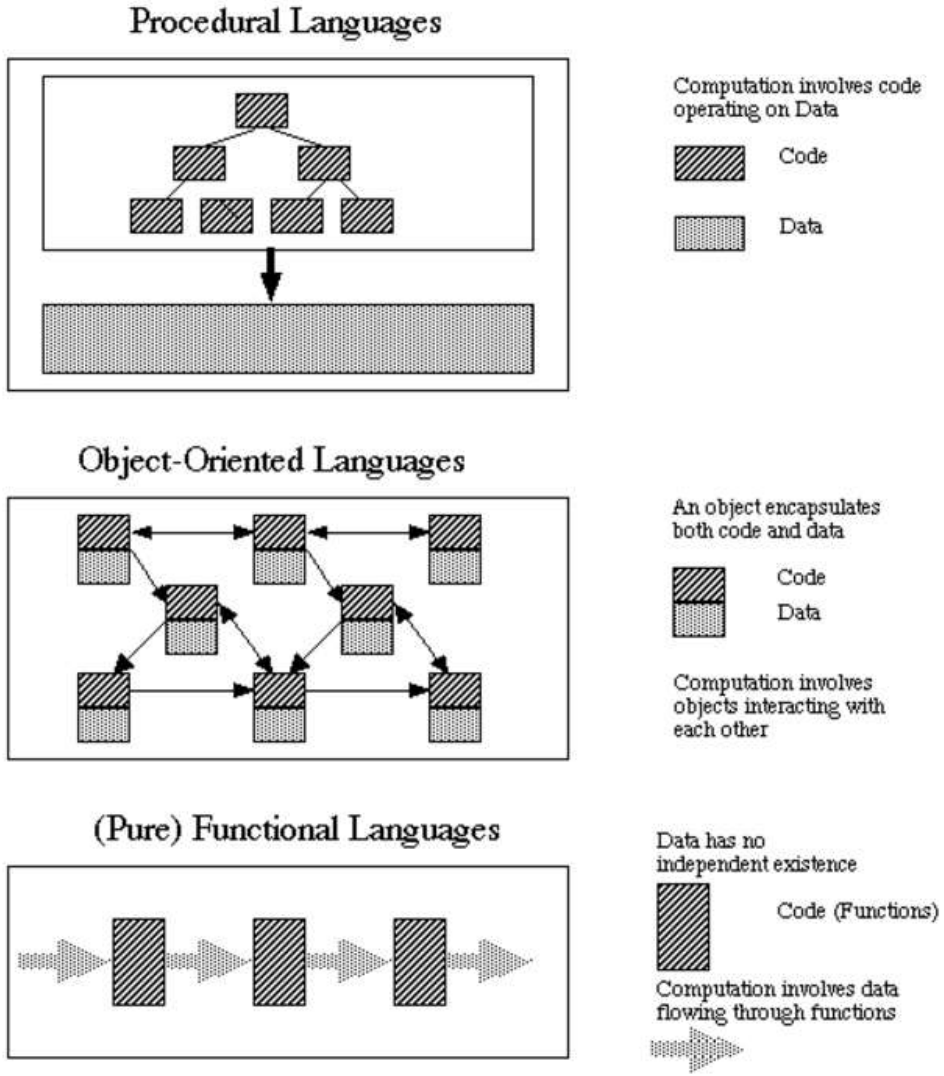
Abstraction

Not mutually exclusive!

Three well-known paradigms:

- **Procedural e.g. C**
- **OOP: Object Oriented Programming e.g. Java, C#**
- **FP: Functional Programming e.g. Haskell, FSharp**

- Reusability
  - Write once use many times
- Modularity
  - Hide information - inside functions
  - Hide information - inside modules
  - Support Abstract Data Types (e.g. queues, lists)
  - **OOP: encapsulate behaviour (state change) with state**
- Composability (make it easy to combine small parts)
  - Reduce side effects
  - Reduce global state
  - **FP: Reduce local state**
  - **FP: Allow functions as data**
  - **OOP: Support making specific classes from general ones**



Type Systems

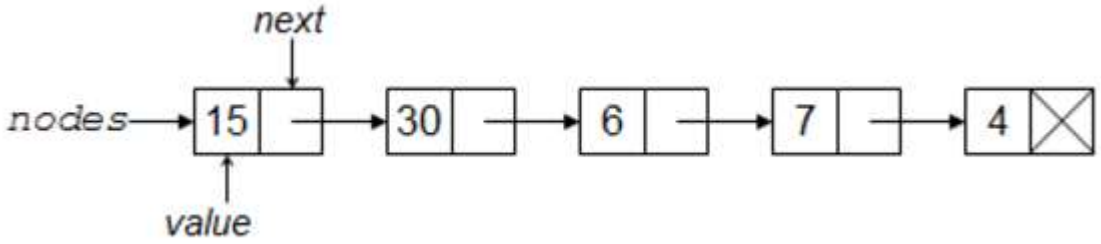
45 min very accessible talk from developer: "What's wrong with Java's Type System"

- Rich type system (Java, C#, F#)
  - Have lots of different types
  - Use types to differentiate and encapsulate data
  - Make it easy for types to model functions and data precisely
- Static vs dynamic type system
  - **Static**: (C, C++, Java, C#, F#) catch all type errors in compiler before running program
    - Types may restrict what you can do
  - **Dynamic** (Python, Matlab, VB): run-time catches type errors when you run program
    - Less protection against bugs, less documentation
    - More flexible?
    - Less type annotation "noise"
  - **Type inference: compiler works out most types from context!**
    - **Rescues static type system from type annotation "noise"**

"Now! *That* should clear up a few things around here!"

A rich, static, type system (without type inference)
credit Dr. Dobbs

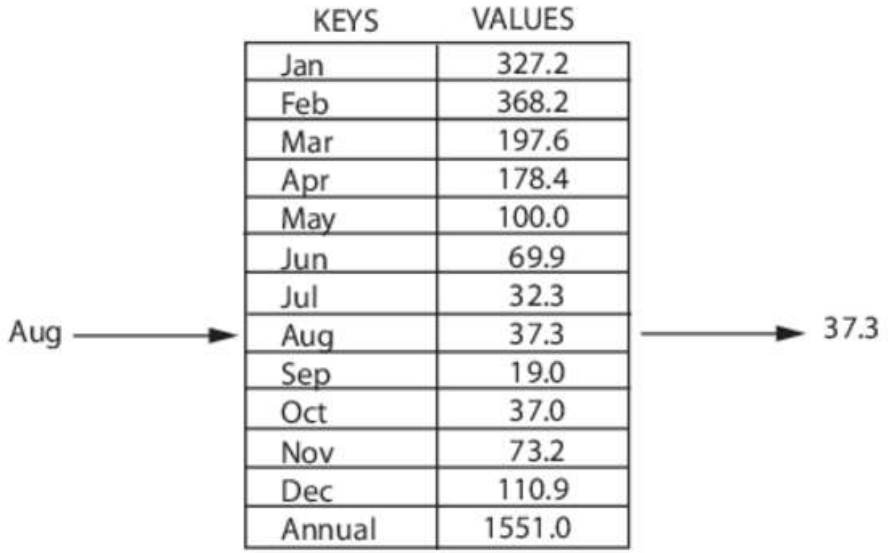
Data Structures



- Good range of built-in data structures
  - List — immutable, use when operating on a collection of data. Note the extra bytes used for pointer
  - Array — mutable, efficient for data storage
  - Map (lookup table)
  - Set — immutable, unordered. Just a container for items. Do not allow duplicated items to be inserted

x = [15 ; 30 ; 6 ; 7 ; 4]

- High level memory management
  - **No pointers needed**
  - **Compiler allocates memory**
  - **Run-time system frees memory (garbage collection)**
  - **Memory allocation errors not possible!**



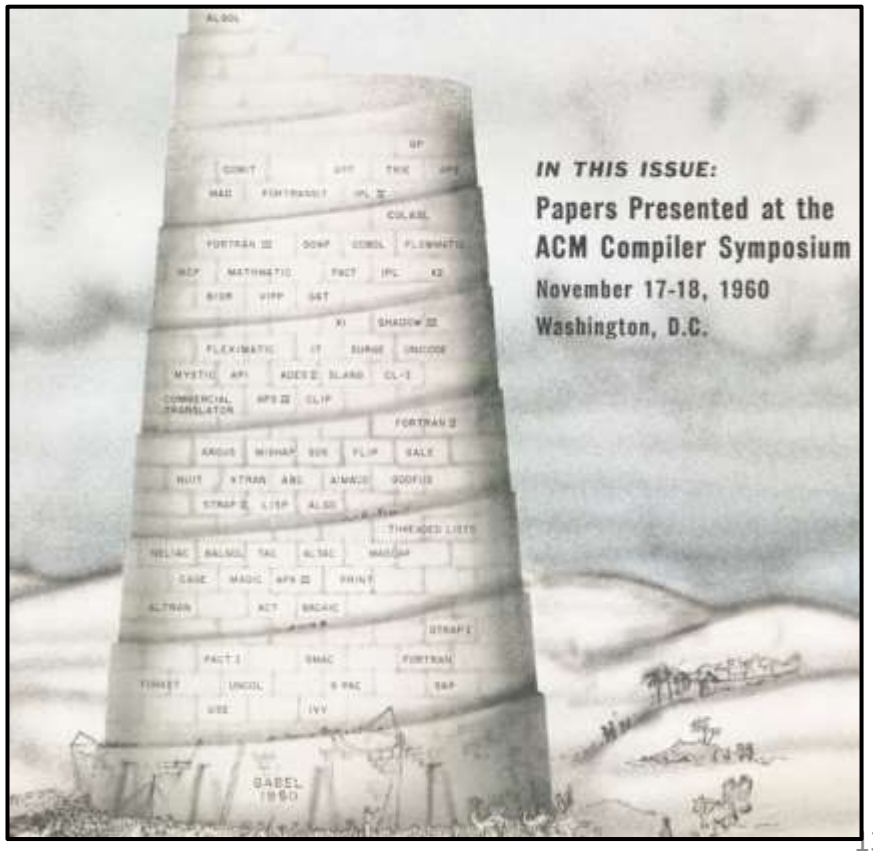
Why learn F#?

- Big question - most of this module will be answering it!
- Short answer: functional languages give you a new perspective on programming **in any language**
  - Improve programming style
  - Learn new concepts
- What is F#?

  - A real functional programming language used in industry with (highly paid) jobs
  - Fully developed and with excellent tooling
  - A practical language. Hybrid functional/OOP. Uses all of .NET libraries and ecosystem.

Almost as soon as people start talking about programming languages they also start criticizing the seemingly bewildering array and variety of such languages. The "Tower of Babel" metaphor was commonly invoked to describe this profusion of programming languages

The iconic reference to the Tower of Babel problem is the cover of the January 1961 issue of the Communications of the ACM, which featured the tower on the cover.



What is a functional language?

- Imperative programming
  - (1) Define and call functions
  - (2) Change the value of variables
- What happens if we get rid of (2)?
- Loops don't work!
  - get rid of them too!
  - use recursion instead of looping
- Programming looks like maths
  - What you want, not how you make it

Imperative

```
int factorial(n) =
{
    int result = 1;
    int i = 1;
    while ( i < n) {
        result = result * i;
        i = i + 1;
    }
     return result;
}
```

Functional

```
let factorial n =
    let numbs = listOfNumbersFrom1ToN
  combineWithMultiply numbs
```

Functional vs Imperative

- Imperative solution makes bugs easy!
  - Order matters
  - Loop limits easy to get wrong
- Functional solution requires new (old?) way of thinking about problem
- Functional solution requires complex data structures
- Functional is simpler

Imperative

```
int factorial(n) =
{
   int result = 1;
   int i = 1;
   while ( i < n) {
      result = result * i
      i = i + 1;
   }
    return result;
}
```

Bug

Order matters

Types are inferred by compiler
factorial: int -> int
numbs: int list

Functional

```
let factorial (n) =
   let numbs = listOfNumbersFrom1ToN
   combineWithMultiply (numbs)
```

Use let to emphasise constant definition not variable

function is always one expression
"return" not needed

F# syntax: unfamiliar!

- Worksheet 1 will introduce new syntax as on right and much more
- Features:
  - `{ }` not used, blocks marked by indentation
  - `;` not needed
  - Function application no longer needs brackets
  - Shorthand for list of numbers
  - `x |> f`
    - `|>` is pipeline operator
    - feeds operand `x` to function `f`
    - same as `f x`
  - `List.reduce (*)`
    - *combineWithMultiply*

Functional

```
let factorial (n) =
    let numbs = listOfNumbersFrom1ToN
    combineWithMultiply (numbs)
```

F#

```
let factorial n =
    let numbs = [1..n]
    numbs |> List.reduce (*)
```

F# (shortened version)

```
let factorial n = [1..n] |> List.reduce (*)
```

Module goals: learn some fun FP concepts

- How does garbage collection work?
- How can pure functions be written mathematically?
- How can a program with pure functions be implemented?
- How can pure functional programs express side-effects?

Module goals: beyond learning a language

- Programming is much more than coding in a language:
  - **Designing programs**
  - **Testing programs**
  - **Controlling dependency**
  - **Modularising & specifying interfaces**
- What makes a programming language good?
  - Static typing (contentious – but should not be!)
  - Functional vs Object Oriented type systems
  - Tooling
  - Libraries & re-usable code base
  - Support for concurrency

you as a developer do not have to specify types every single time

theoretical issues
versus
pragmatic issues

Discover how the language and type system affects code readability and reusability

Learn how to write concurrent programs (the easy way)

Questions?