Aston University
BIRMINGHAM UK

Introduction

CS2PLC Programming Language Concepts

Module Instructors

Hassan Aqeel Khan
h.khan54@aston.ac.uk
Office MB211K

Michael Pritchard
m.pritchard2@aston.ac.uk
Office MB209

Module Evaluation

Assessment Type	Category	Duration/ Submission Date	Common Modules/ Exempt from Anonymous Marking	Assessment Weight
Quiz	**Open Book**	**TBC**	**Yes**	**40%**
Details	Blackboard quizzes based on practical tasks, during, select, 2-hour practical sessions. There are 3 quizzes. The best 2 quizzes count. The practical tasks take approximately 90 minutes and the quiz approximately 30 minutes.			
February to June Exam	**Closed Book**	**1:30hrs**	**-**	**60%**
Details	Computer based, in a computer room on campus.			
			Total:	**100%**

Aston University
BIRMINGHAM UK

Recommended Reading

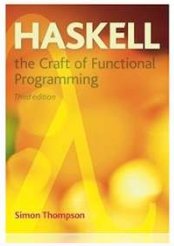
Programming language design concepts

Book - by David A. Watt - 2004 - **Recommended**

Programming language design concepts

Book - by David A. Watt; William Findlay; ProQuest (Firm) - c2004 - **Recommended**

Ebook Central



Haskell: the Craft of Functional Programming (3rd Edition) by Simon Thompson
Available for free download at: https://www.haskellcraft.com/craft3e/Buy.html

Outcomes Summary

- Motivation: Why study PL concepts
- What a PL is for and how different paradigms achieve it
- What makes a good PL and why there is no perfect PL

Module Arrangements - Schedule

- Independent learning with asynchronous support
  - Books, notes, recordings
  - Code on GitHub

  2-3 hrs
  - Practical tasks
  - Wiki with links to online tutorials etc
  - BB discussion forum

  4-5 hrs
- Thursday lecture sessions
  - Reflection on past assessed/self-assessed work
  - Preparation for imminent assessed work
  - Weeks' unit(s)

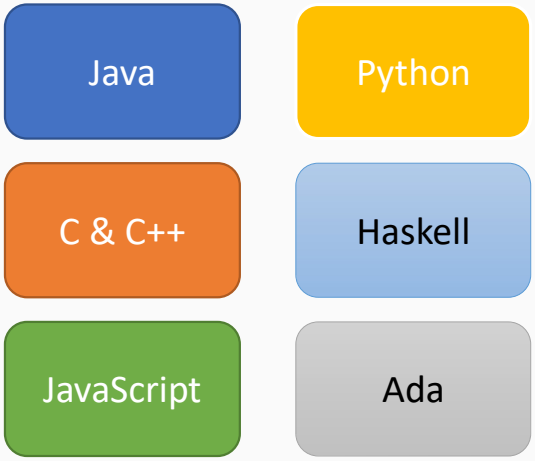
  2 hrs
- Assessed practical sessions
  - Working on practical tasks in self-selected teams of 3-4
  - **Individually-assessed** quiz related to practical tasks
  - Lab Quizzes in weeks commencing **8-Feb, 01-Mar, 22-Mar**

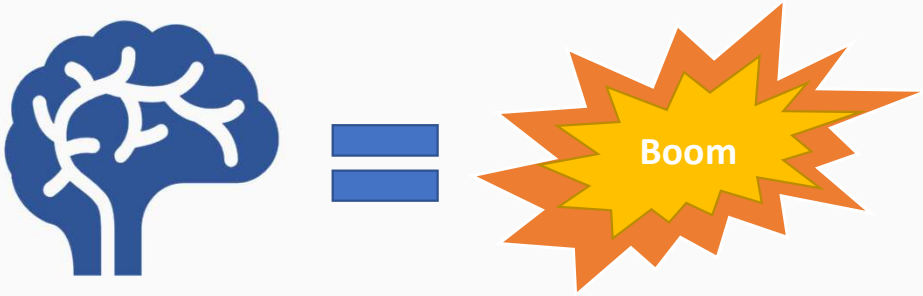
  2 hrs

10-12 hrs per week

Why Study PLC

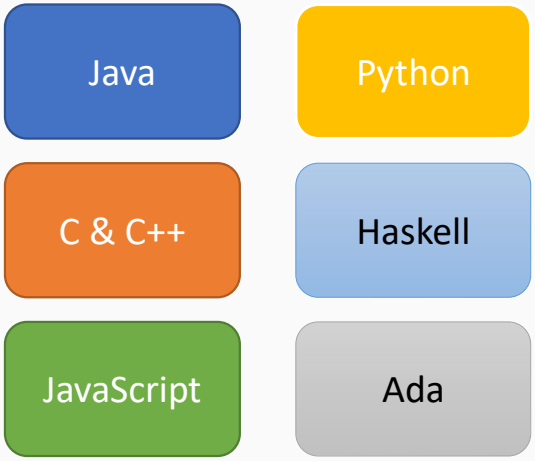
- PLC will help you reflect on what you already know (i.e. Java), **conceptualize** the most important aspects and compare them with **several** other major PLs.
- Here are some languages that you will encounter in PLC





Why Study PLC

- PLC will help you reflect on what you already know (i.e. Java), **conceptualize** the most important aspects and compare them with **several** other major PLs.
- Here are some languages that you will encounter in PLC



RELAX! PLC focuses more on **semantic** issues than **syntax**

Why Study PLC

- Increased capacity to express programming ideas
- Increased ability to learn new programming languages
  - Sorry programmers but you don't have the luxury of comfort zones
- Improved background for choosing a programming language
- Increased ability to design good quality new languages
- Overall advancement of computing and an improvement in the quality of software
  - Yes, you can make the world a better place (for other programmers at least)

Assessed Outcomes

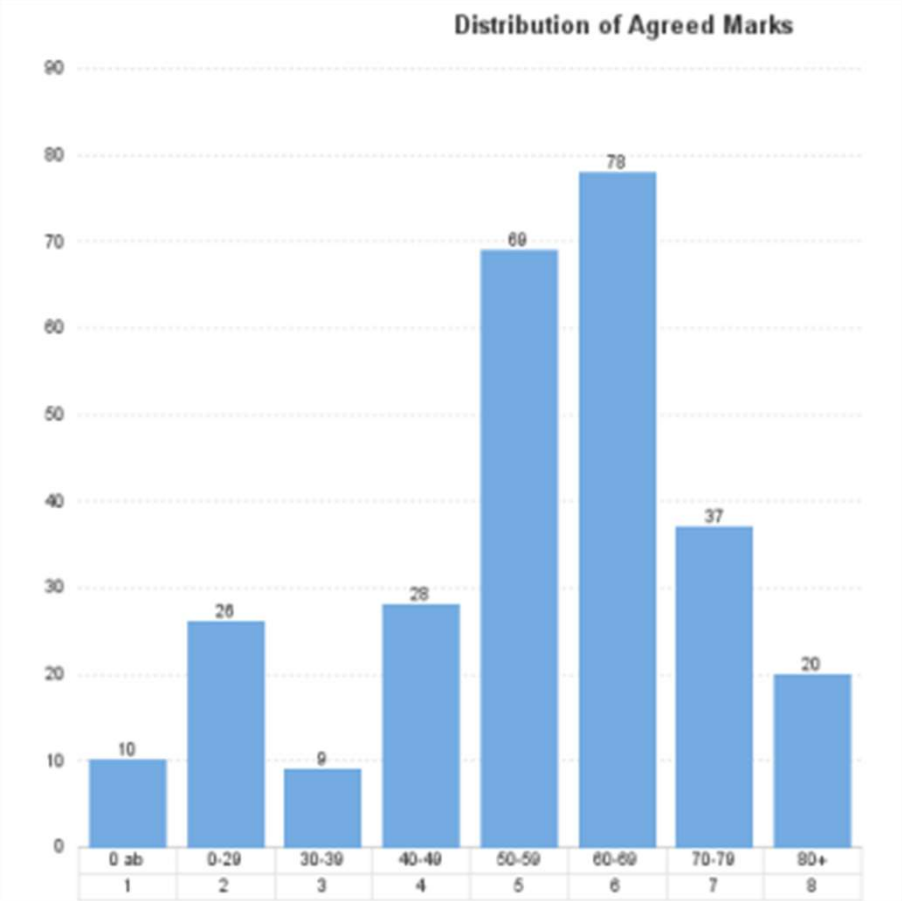
You will demonstrate sufficient understanding of PLC by being able to:

- Critically evaluate and analyse the suitability of a programming language appropriate for a given task by demonstrating awareness of the PL choices available, and knowledge of the relative strengths and weaknesses of commonly used languages and their selected features.
- Exhibit working knowledge of multiple programming languages by implementing, analysing, or modifying small programmes.
- Demonstrate knowledge and practical ability by completing simple tasks based on understanding of demo-code and documentation related to key aspects of the programming languages.

Assessed Outcomes

- How to achieve these skills?
  - By your work in practicals and in-lecture exercises
  - It is recommended that you help each other while learning via group work and using the official discussion forum.
  - If other methods fail, you are welcome to email the lecturer or arrange a meeting via WASS.

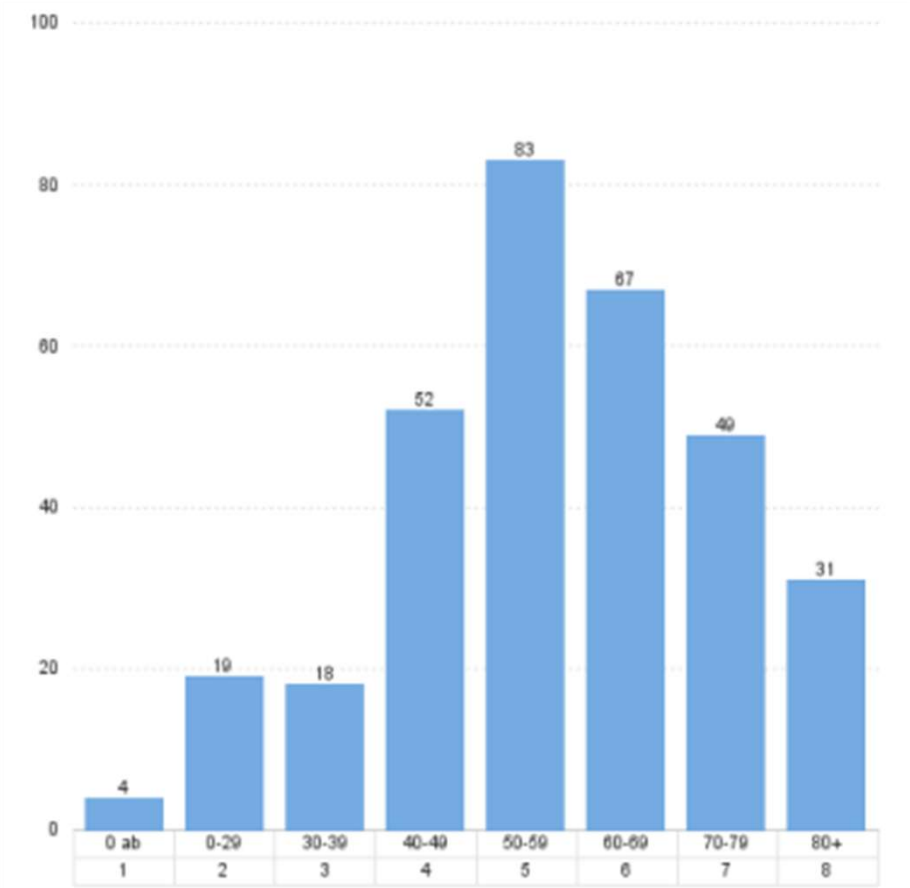
Module Results 2022-23



Calculated Marks	All Students	Excl All Absences
Students	277	259
Pass Count	232	232
Lowest	0	0.67
Highest	93.4	93.4
Mean Mark %	55.06	58.89
Standard Deviation	19.57	16.16
Overall Pass Rate %	83.75	89.58

Aston University
BIRMINGHAM UK

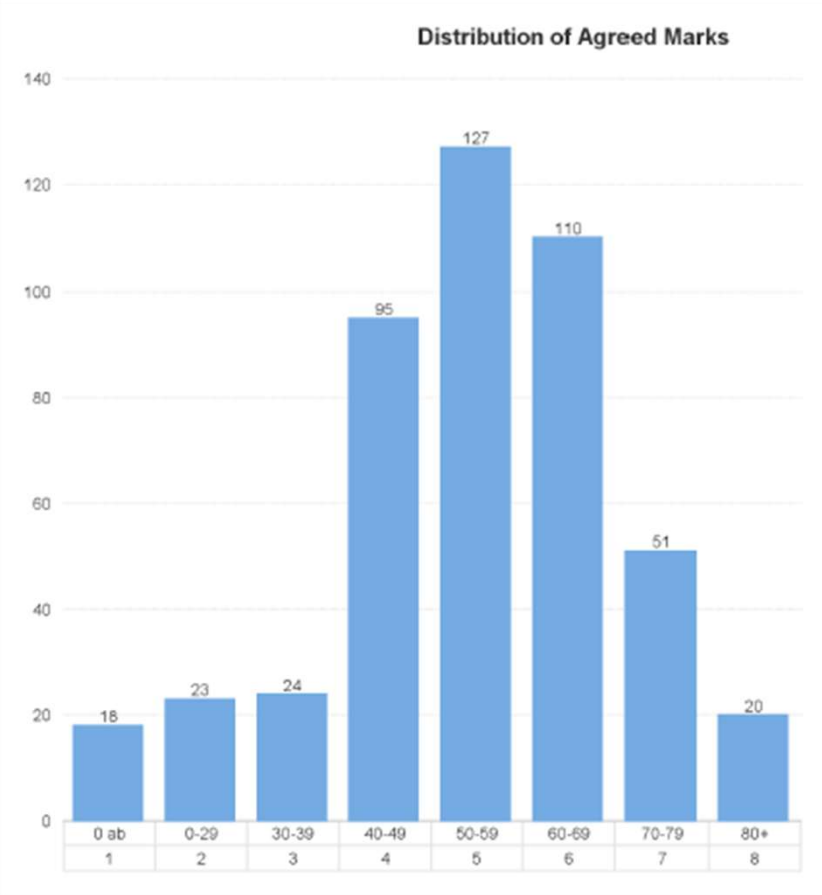
Module Results 2023-24



Calculated Marks	All Students	Excl All Absences
Students	323	317
Pass Count	282	282
Lowest	0	1
Highest	94.2	94.2
Mean Mark %	57.08	58.16
Standard Deviation	17.72	17.46
Overall Pass Rate %	87.31	88.96

Aston University
BIRMINGHAM UK

Module Results 2024-25



Calculated Marks	All Students	Excl All Absences
Students	468	450
Pass Count	403	403
Lowest	0	3
Highest	92.5	92.5
Mean Mark %	53.28	55.41
Standard Deviation	17.53	15.47
Overall Pass Rate %	86.11	89.56

Aston University
BIRMINGHAM UK

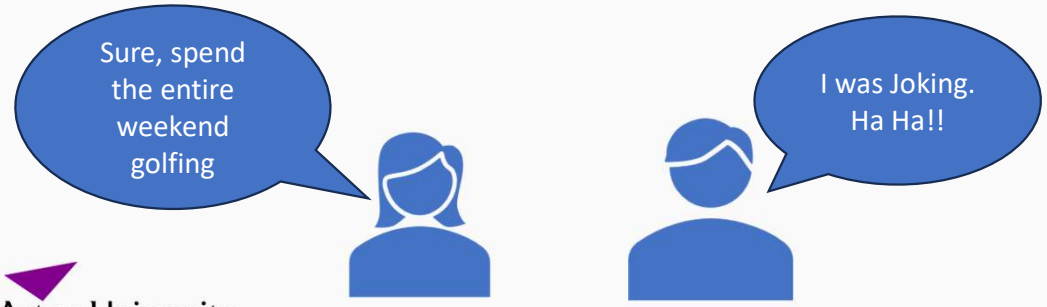
Natural Languages Vs Programming Language

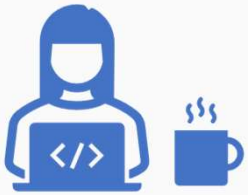
Natural Languages

- Target audience: Humans only
- Can have multiple connotations
- Acquired over time via informal training
- Helps if audience understands underlying cultural context

Programming Languages

- Target audience: Man & Machine
- Multiple connotations = Confusion
- Acquired over time via formal training (and lots of coffee)
- Helps if programmer understands underlying design process (and/or computer architecture)





Aston University
BIRMINGHAM UK

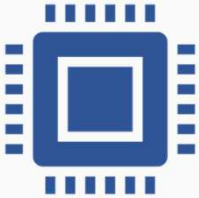
Purpose of PLs

- Make a computer do something
- PL should be easy to understand, for both human and computer

English
French
Spanish
Chinese

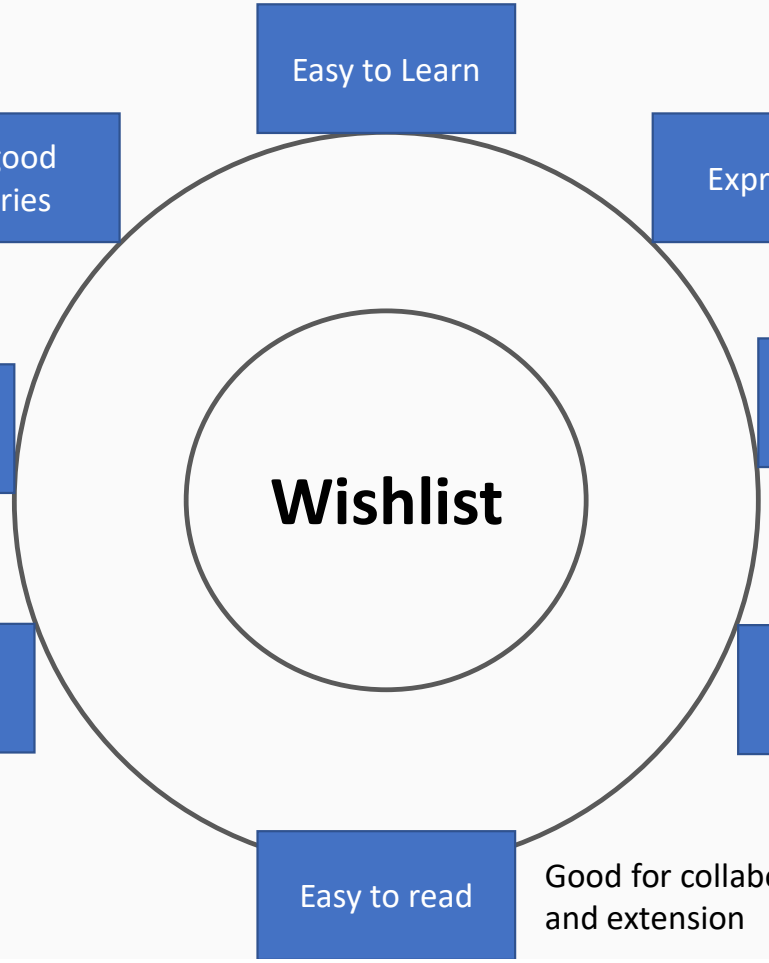
```
// Your First Program

class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
mov rax, 1
mov rdi, 1
mov rsi, message
syscall
```



Aston University
BIRMINGHAM UK

The Perfect PL



If I have seen further than others, it is by standing upon the shoulders of giants

Get more with less code

with great processing power comes great data to process

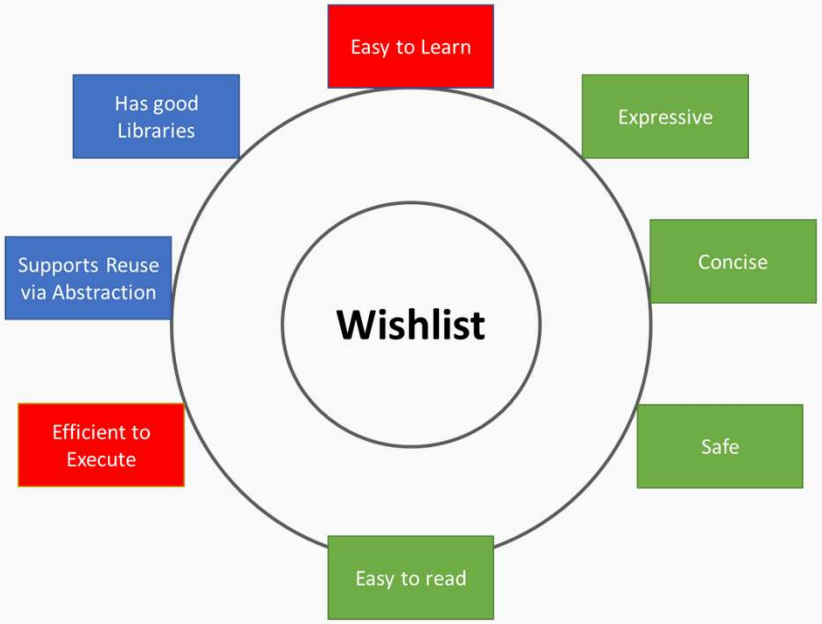
keep an eye out for human screw-ups

Good for collaboration and extension

The Perfect PL DOES NOT Exist

- Some of the criteria are contradictory

  For example, concise and expressive PLs are typically abstract and thus difficult to learn. They also suffer from efficiency problems. Examples: Lisp, Haskell, and Prolog.



The Perfect PL DOES NOT Exist

- Many of the criteria are context dependent
  - A language can be expressive in certain application areas but very cumbersome for use in other areas.
  - Pretty much all the mainstream languages are optimised for the specific application domains where they originated

Java: Web and Internet programming

JavaScript: dynamic Web page behaviour

C: operating system programming, device drivers

Ada: military safety-critical applications

Prolog and Lisp: AI, rule-based systems

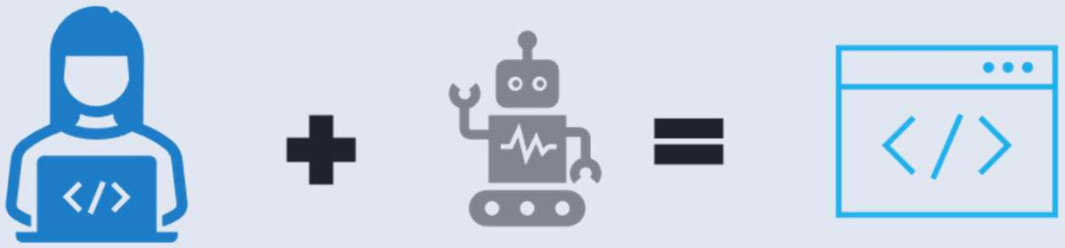
Haskell: academic playground for new PLC ideas

Some Very Recent Developments

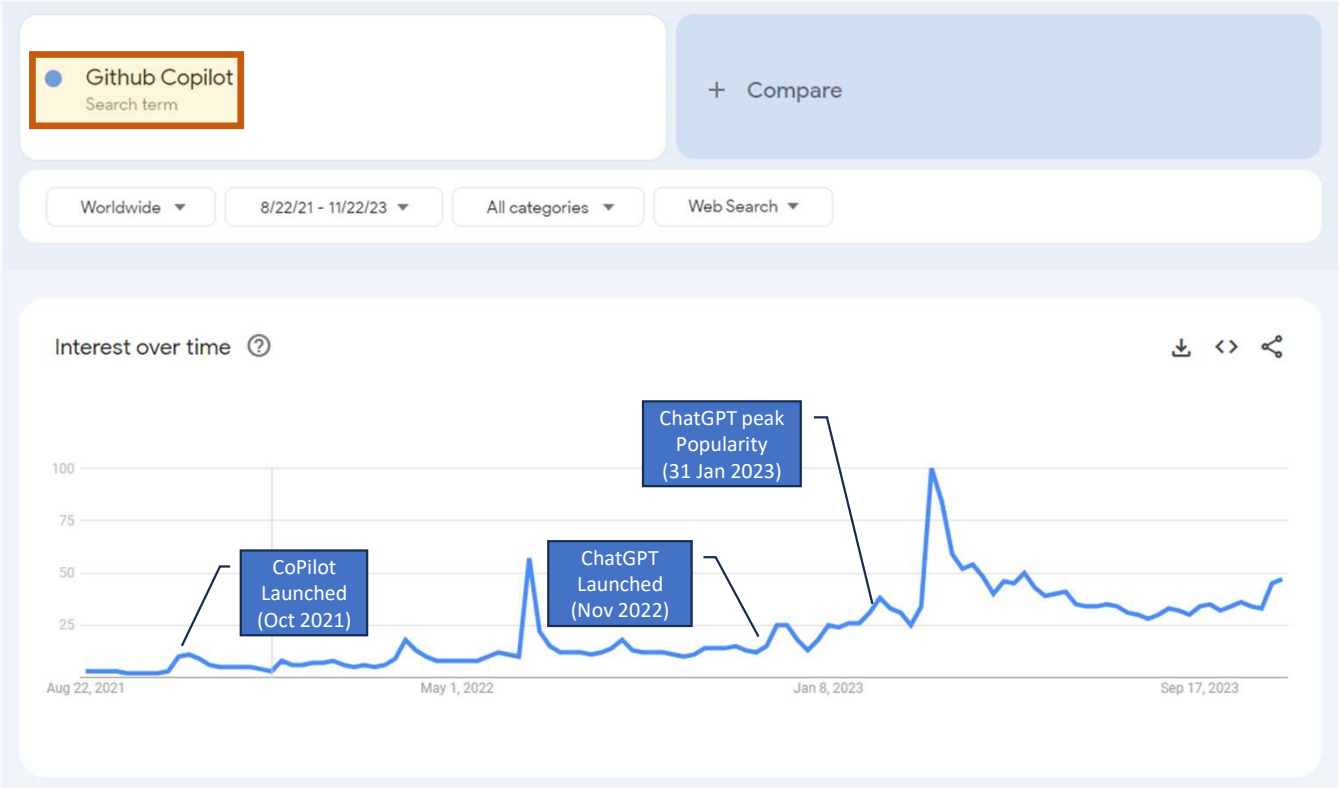
- Generative AI based Coding Tools
- How useful are they?
- What impact are they expected to have on the Software Development job market?

What are GenAI Coding Tools?

- Software assistants that use artificial intelligence (AI) and natural language processing (NLP) to help programmers code faster and with higher accuracy.
- Popular Tools:
  - Tabnine (Launched 2018)
  - GitHub CoPilot (Launched Oct 2021)
  - Amazon CodeWhisperer (Launched June 2022)
  - Divi AI (Launched Aug 2023)
  - Replit AI (Launched Oct 2023)



Impact of the ChatGPT Boom

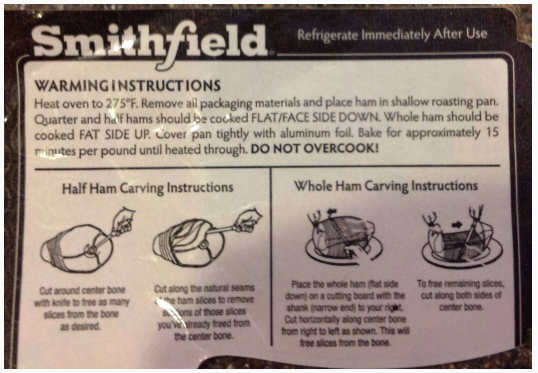


Source: Google Trends

Main Programming Paradigms

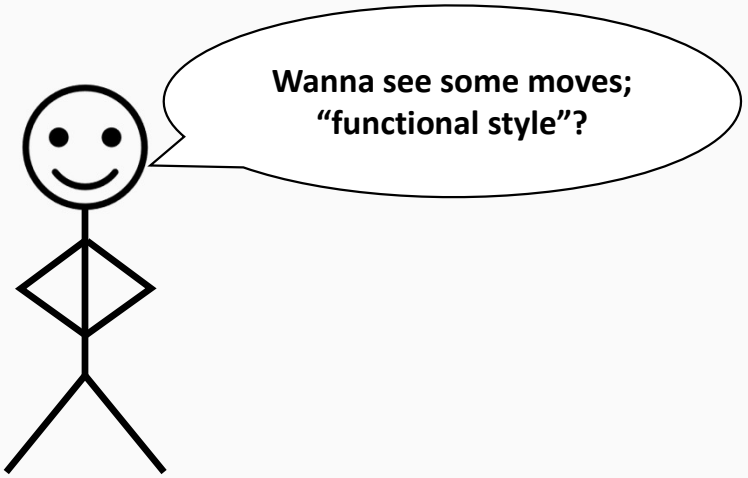
Imperative PLs

- Most common high-level PLs fall within this paradigm
- View computation as structured data stored in memory along with step-by-step instructions to manipulate the data
- **Essence:** "Tell the computer what to do step-by-step"
- **Sub-Categories:** Procedural and Object-Oriented PLs
- Examples: Java, C, Ada, JavaScript, PHP and Python



Functional PLs

- Functional Programming:
  - *Style* of programming in which the basic method of computation is the application of functions to arguments.
- A **functional PL** is one that supports and encourages the *functional style*



Thinking Functionally

An Intro to the CORE Concepts of Functional Programming

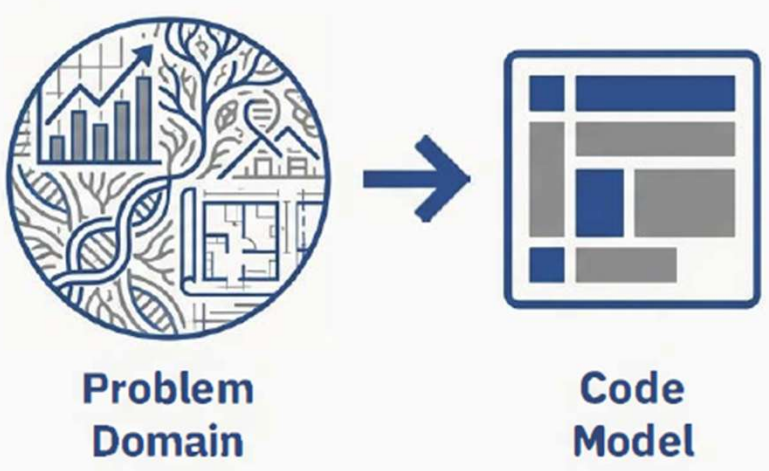
Aston University
BIRMINGHAM UK

All programming is modelling

The fundamental purpose of computing is to process and manipulate symbolic information.

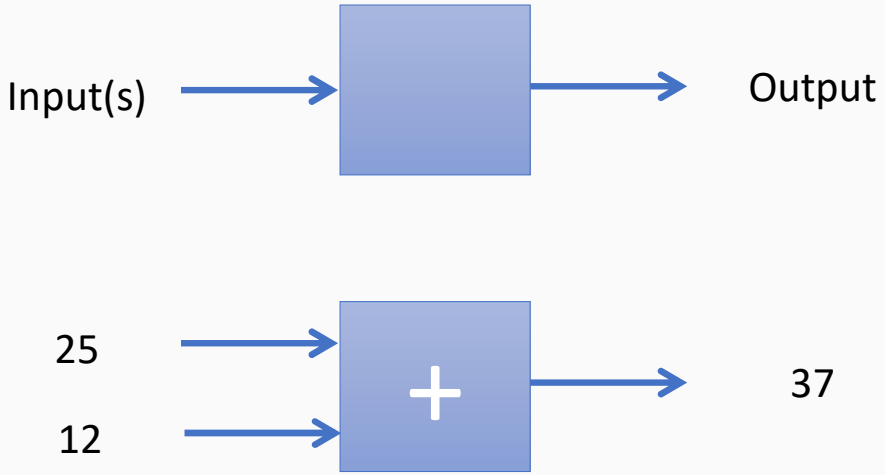
A program is a description of how this information is manipulated.

Functional programming offers a powerful and direct way to model situations-real-world or imaginary and the rules that govern them, using a few simple, core ideas.



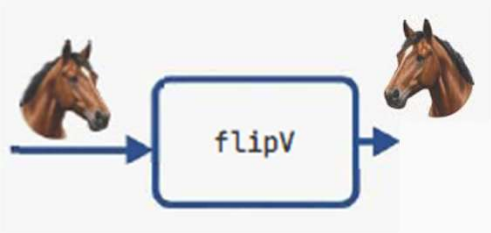
The Fundamental Tool: The Function

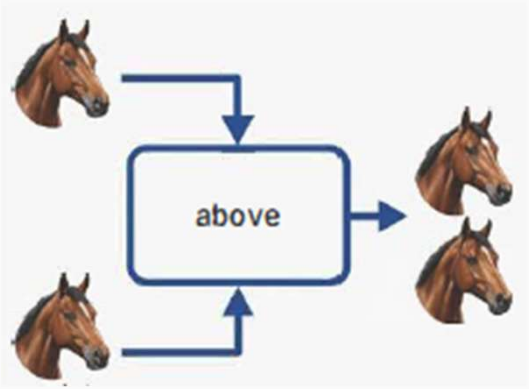
A function is a predictable transformation. It takes one or more inputs (also called arguments or parameters) and produces a single output (or result). The same input will *always* produce the same output. The process of giving inputs to a function is called *function application.*



Functions in Action: Modeling Pictures

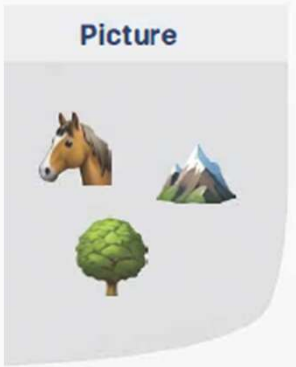
We can model common relationships between pictures using functions. These functions become our basic building blocks for creating more complex images.

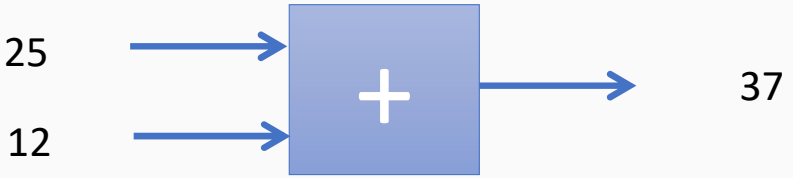




Types Bring Order to Our Values

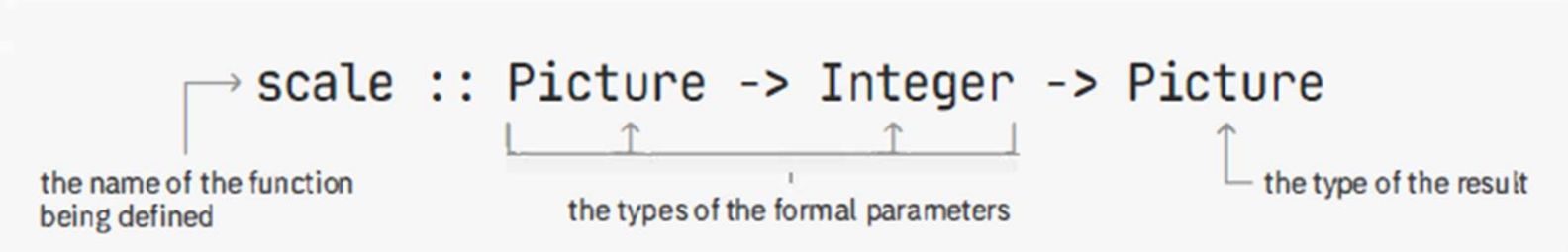
- A **type** is a name for a collection of related values. It tells us what *sort* of thing we are working with.
- For example, 5 is an `Integer`, while the image of a horse is a `Picture`. Giving values a type.
- Giving values a type helps us avoid nonsensical operations, like trying to add a number to a picture.





Typed Functions: A Blueprint for Behavior

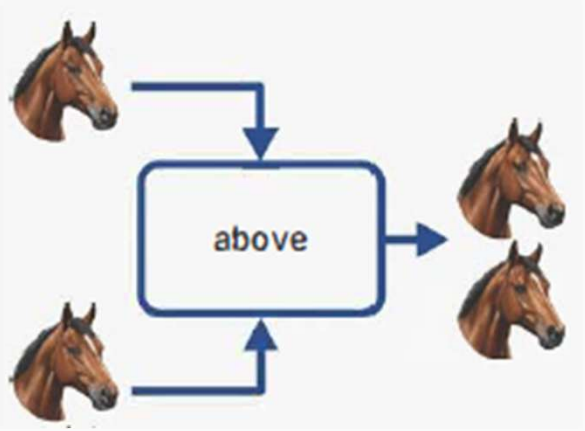
A function's **type signature** is its **contract**. It precisely describes the type of inputs the function expects and the type of output it will produce. This tells us not only how functions can be used but also checks that they are being used in the right way.



The Type System is Your Safety Net

The type system automatically checks these contracts.

Exercise: Write the type signature for the above function shown in the figure



Answer

```
above :: Picture -> Picture -> Picture
```

Name of function

types of Formal Parameters

type of the Result

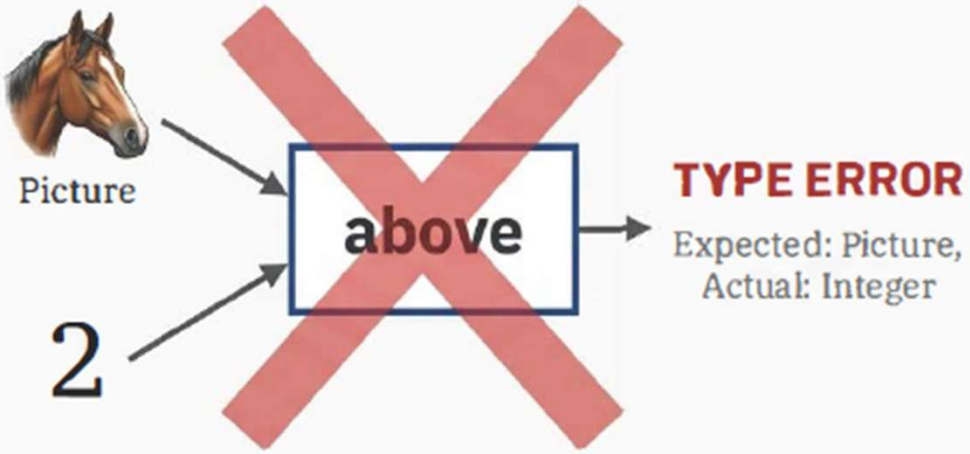
The Type System is Your Safety Net

The type system automatically checks these contracts.

Applying a function to the wrong type of data results in a **type error.**

This is not a crash at runtime; it's a predictable failure during compilation, allowing you to fix the mistake immediately.

Type checking helps to write correct programs and avoid a large proportion of programming pitfalls.



Computation is Step-by-Step Calculation

A program runs by *evaluating expressions* to find their final value. This is a transparent and predictable process of simplification, just like solving '(7-3) * 2' becomes '4 * 2' which becomes '8'. In functional programming, we do exactly the same, but our expressions use functions.

```
invertColour (flipV horse)
```

- **Note** that here we are passing a function as an argument to another function.
- The function `flipV`, with input argument `horse`, is being passed to the function `invertColour`

Computation is Step-by-Step Calculation

A program runs by *evaluating expressions* to find their final value. This is a transparent and predictable process of simplification, just like solving '(7-3) * 2' becomes '4 * 2' which becomes '8'. In functional programming, we do exactly the same, but our expressions use functions.

```
invertColour (flipV horse)
```

evaluation

```
invertColour (flippedhorse)
```

evaluation

```
whitehorse
```

We Build Our Vocabulary with Definitions

- A functional program consists of a number of *definitions.* We can give a name (an identifier) to a value or to the result of an expression.
- By convention,
  - names for functions & values start with a **small** letter (e.g size, blackHorse)
  - type names start with a **capital** letter (e.g. Picture, Int, Num)

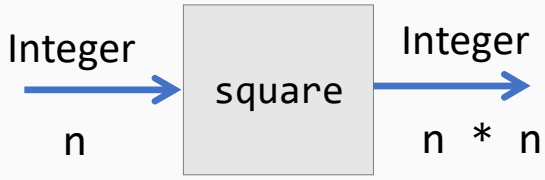
```
-- Defining a value with its type
size :: Integer
size = 12 + 13
```

```
-- Defining a picture from an expression
blackHorse :: Picture
blackHorse = invertColour horse
```

Defining Our Own Function "Machines"

- We define our own functions in two parts. First, we declare the type of the thing being defined (the type signature*). Second, we provide an equation that says when the function is applied to an unknown value or variable (like n), the result is given by an expression.

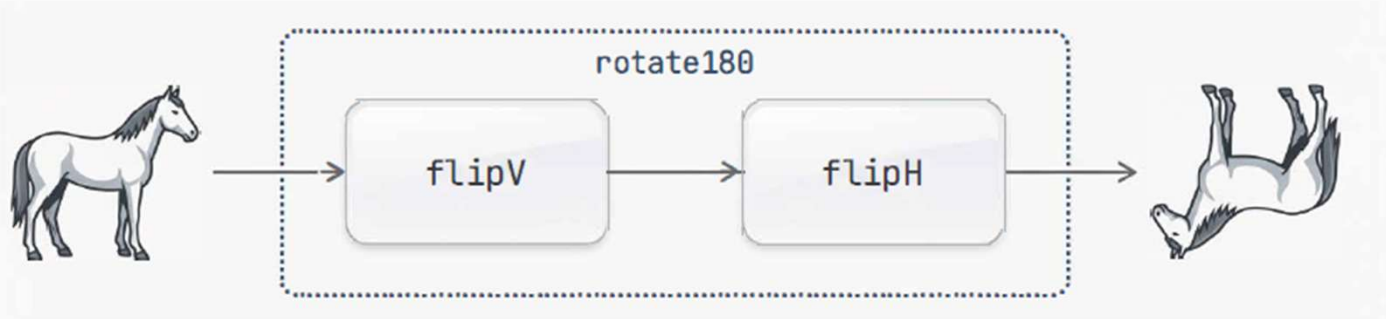
```
square :: Integer -> Integer
square n = n * n
```



* Strictly speaking, writing type signature for your functions is NOT mandatory and the Haskell compiler will infer it if you don't write it; however, writing type signature is considered good practice as it adds a useful layers of documentation and allows code users to verify the intent of the function's creator.

Composition: Chaining Our Machines Together

The output of one function can become the input of another. This powerful idea is called ***function composition***. It is so common that Haskell provides a special operator (.) for it. This lets us build sophisticated operations from simple, reusable parts.



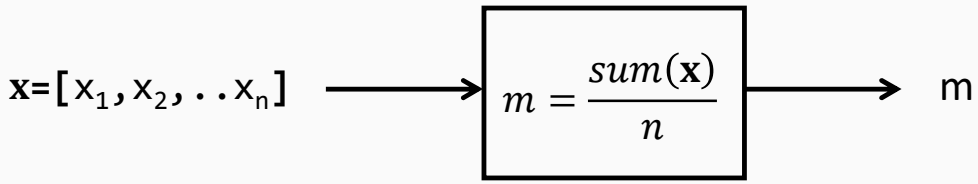

```
-- A standard definition for rotation
rotate180pic :: Picture -> Picture
rotate180pic pic = flipH (flipV pic)
```

```
-- The same function, defined using composition
rotate180 :: Picture -> Picture
rotate180 = flipH . flipV
```

Exercise: Writing type signatures for Haskell Functions

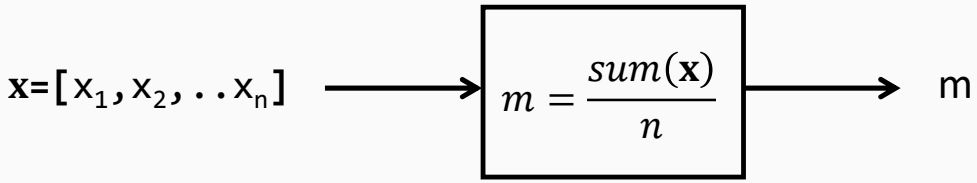
What will be the type signature of a function called `mean` that calculates the mean value of the numbers in a list?

Let's first think about what this function does.



Exercise: Writing type signatures for Haskell Functions

What will be the type signature of a function called `mean` that calculates the mean value of the numbers in a list?



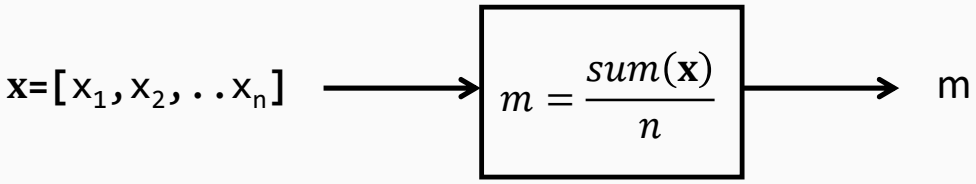

```
mean ::  list of numbers -> type of m
```

name of function

types of Formal Parameters

type of the Result

Exercise: Writing type signatures for Haskell Functions



The Five Numeric Types in the Prelude	
Integral	`Int` (fixed-precision)
	`Integer` (unlimited-precision)

Fractional	`Float`
	`Double`
	`Rational` (no rounding error)

```
mean ::  list of numbers -> type of m
```

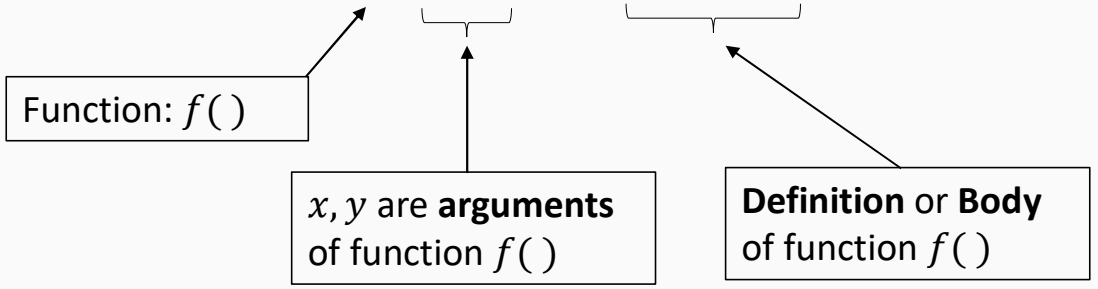
More formally,

```
mean :: [Double] -> Double
```

Background: How to Write a Function in Haskell

- Haskell is a functional PL
- Syntax-wise, writing functions in Haskell is different from what you are used to (in Java)
- However, the syntax of writing a function in Haskell is similar to writing a mathematical function
- Mathematical Function:

$$f(x, y) = x^2 + y^2$$



Background: How to Write a Function in Haskell

- Mathematical Function: $f(x, y) = x^2 + y^2$

Function: $f(\,)$

x, y are **arguments** of function $f(\,)$

Definition or **Body** of function $f(\,)$

Haskell (Functional PL)

```
func x y = x^2 + y^2
```

Function: `func`

`x, y` are **arguments** of function `func`

Definition or **Body** of function `func`

*NOTE: Functions are typically referred to as 'Methods' in Java

Background: How to Write a Function in Haskell

- Mathematical Function: $f(x, y) = x^2 + y^2$

Haskell (Functional PL)

```
func x y = x^2 + y^2
```

Equivalent Java (Imperative PL) Method/Function

```
private static double
        func(double x, double y)
        {
        return (Math.pow(x,2) + Math.pow(y,2));
        }
```

Imperative Vs Functional PLs

- Let's consider a simple summation function.

$$y = \sum_{x=0}^{5} x$$

Java implementation	Haskell Implementation

`int total = 0;`
`for (int i = 0; i <= 5; i++)`
`total = total + i;`
	`sum [0..5]`

Imperative Vs Functional PLs

- Let's consider a simple summation function.

$$y = \sum_{x=0}^{5} x$$

There are Two functions* here

Java implementation

```
int total = 0;
for (int i = 0; i <= 5; i++)
        total = total + i;
```

Haskell Implementation

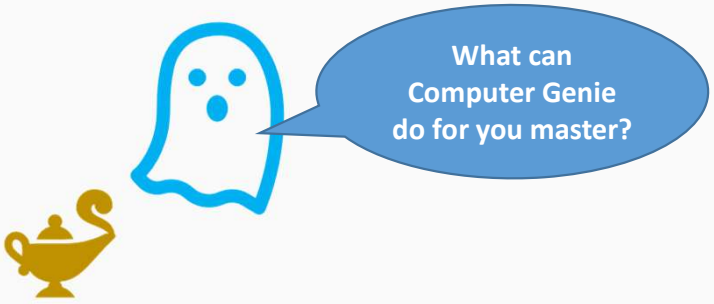
```
sum [0..5]
```

Function-1: **[..]** is a library function used to generate a list of integers from 0 to 5

Function-2: **sum** is a library function used to sum the elements of the list.

Functional PLs

- View computation as the evaluation of **mathematical** functions.
- **Essence:** Do not give the computer step-by-step instructions, rather "Describe what it is you want from it".
- Basic method of computation is application of functions to arguments.
- One advantage of pure functional PLs over imperative PLs is no **side-effects**.
- **Reward:** Simpler code behaviour leading to better maintainability, reusability and reliability.



Functional PLs

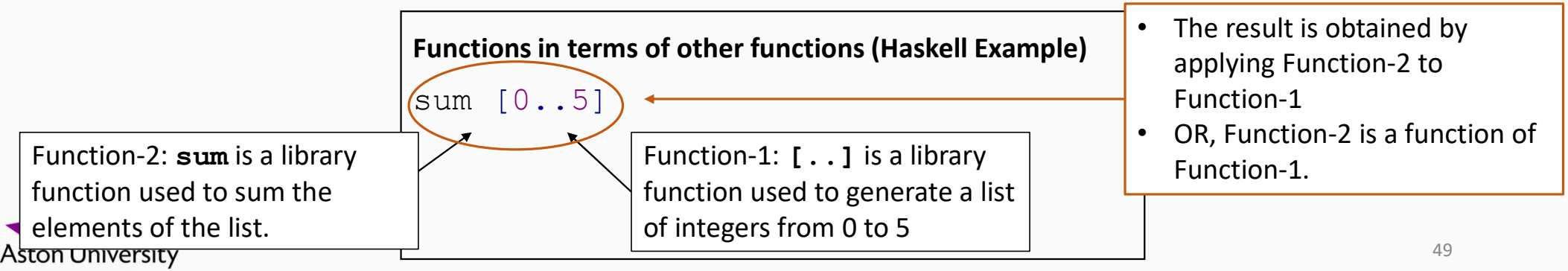
- Functions are first-class citizens [1],
- **Meaning**
  - Functions are no different to other data used in a program
  - Functions can be used as arguments to functions
  - Functions can be returned as values from other functions
  - Function can be written in terms of other functions (just like Math)

Functions in terms of other functions (Math Example)

$$f(x) = \frac{g(x)}{m(x)} \text{ (e.g., } \tan(x) = \frac{\sin(x)}{\cos(x)} \text{)}$$

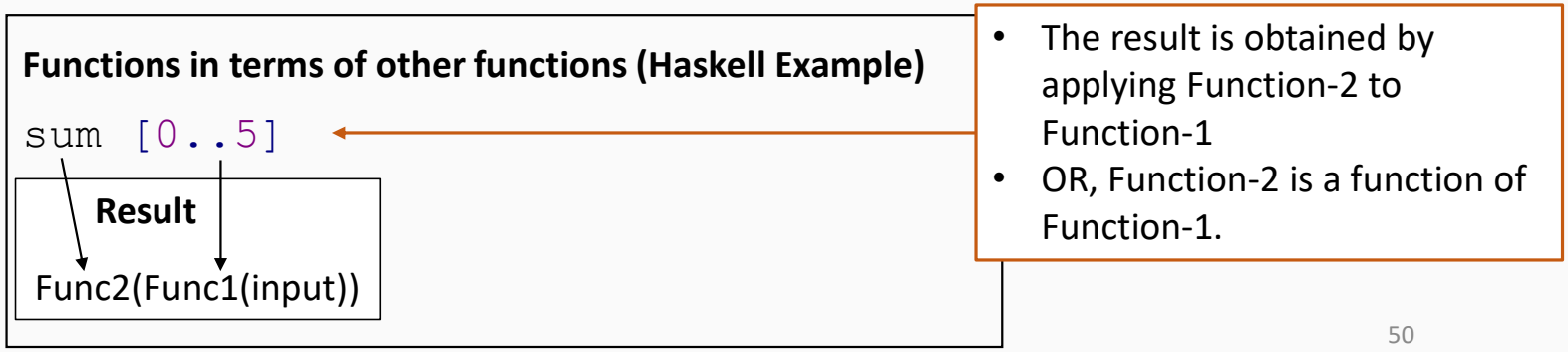
Functional PLs

- Functions are first-class citizens [1],
- **Meaning**
  - Functions are no different to other data used in a program
  - Functions can be used as arguments to functions
  - Functions can be returned as values from other functions
  - Function can be written in terms of other functions (just like Math)



Functional PLs

- Functions are first-class citizens [1],
- **Meaning**
  - Functions are no different to other data used in a program
  - Functions can be used as arguments to functions
  - Functions can be returned as values from other functions
  - Function can be written in terms of other functions (just like Math)



Functional PLs

- **Functions in terms of other functions**
- Sum Function: $f(x) = \sum_{x=1}^{5} x$
- Average Function: $m(x) = \frac{1}{5}\sum_{x=1}^{5} x = \frac{1}{5} \times f(x) = m(f(x))$
- Combined Function: $g(x) = g\big(f(x)\big) = \begin{cases} \sum_{x=1}^{5} x & if\ sum \\ \frac{1}{5}\sum_{x=1}^{5} x & if\ average \end{cases}$

Functions as Arguments

Functional PLs: Easy

Haskell Example

```
inpFunc = sum[1..5]

applicatorFunc inpFunc s = if s=='s' then inpFunc else inpFunc/5


applicatorFunc inpFunc 'a'
Output: 3.0
applicatorFunc inpFunc 's'
Output: 15.0
```

Functions as Arguments

Functional PLs: Easy

Haskell Example

Input Function: $f(x) = \sum_{x=1}^{5} x$

Combined Function: $g(x) = g(f(x)) = \begin{cases} \sum_{x=1}^{5} x = f(x) & if\ sum \\ \frac{1}{5}\sum_{x=1}^{5} x = \frac{1}{5}f(x) & if\ average \end{cases}$

```
inpFunc = sum[1..5]

applicatorFunc inpFunc s = if s=='s' then inpFunc else inpFunc/5


applicatorFunc inpFunc 'a'
Output: 3.0
applicatorFunc inpFunc 's'
Output: 15.0
```

Arguments to `applicatorFunction`

Functions as Arguments

Functional PLs: Easy

Haskell Example

```
inpFunc = [1..5]

applicatorFunc inpFunc s = if s=='s' then sum inpFunc else (sum inpFunc)/5

applicatorFunc inpFunc 'a'
Output: 3.0

applicatorFunc inpFunc 's'
Output: 15.0
```

Functions as Arguments

Imperative PLs: Possible but cumbersome

```
5   import java.util.*;
6   import java.io.*;
7   interface testSum
8   {
9     void set(int maxInt);
10  }
11  class sumInts {
12
13      void applyFunc(testSum fun, int maxInt)
14      {
15        fun.set(maxInt);
16      }
17
18      void sumfunc(int maxInt)
19      {
20        int total = 0;
21        for (int i = 0; i <= maxInt; i++)
22          total += i;
23        System.out.println("Total-> " + total);
24      }
25
```

```
26      void avgfunc(int maxInt)
27      {
28          int total = 0;
29          for (int i = 0; i <= maxInt; i++)
30              total += i;
31          float avg = (float)total/(maxInt+1);
32          System.out.println("Average-> " + avg);
33      }
34
35      public static void main(String[] args)
36      {
37          char selection = 'a';
38          sumInts L=new sumInts();
39          if (selection=='s'){
40              testSum func = L::sumfunc;
41              L.applyFunc(func, 5);
42          }
43          else if (selection=='a'){
44              testSum func = L::avgfunc;
45              L.applyFunc(func, 5);
46          }
47          else{
48              System.out.println("Incorrect selection");
49          }
50          System.out.println("Done");
51      }
52  }
```

Functions as Arguments

Imperative PLs: Possible but cumbersome

```
5   import java.util.*;
6   import java.io.*;
7   interface testSum
8   {
9     void set(int maxInt);
10  }
11  class sumInts {
12
13      void applyFunc(testSum fun, int maxInt)
14      {
15        fun.set(maxInt);
16      }
17
18      void sumfunc(int maxInt)
19      {
20        int total = 0;
21        for (int i = 0; i <= maxInt; i++)
22          total += i;
23        System.out.println("Total-> " + total);
24      }
25
```

```
26      void avgfunc(int maxInt)
27      {
28          int total = 0;
29          for (int i = 0; i <= maxInt; i++)
30              total += i;
31          float avg = (float)total/(maxInt+1);
32          System.out.println("Average-> " + avg);
33      }
34
35      public static void main(String[] args)
36      {
37          char selection = 'a';
38          sumInts L=new sumInts();
39          if (selection=='s'){
40              testSum func = L::sumfunc;
41              L.applyFunc(func, 5);
42          }
43          else if (selection=='a'){
44              testSum func = L::avgfunc;
45              L.applyFunc(func, 5);
46          }
47          else{
48              System.out.println("Incorrect selection");
49          }
50          System.out.println("Done");
51      }
52  }
```

Explanation: I'm not even gonna dare! 😄

Functional PLs – Haskell mini demo (1/2)

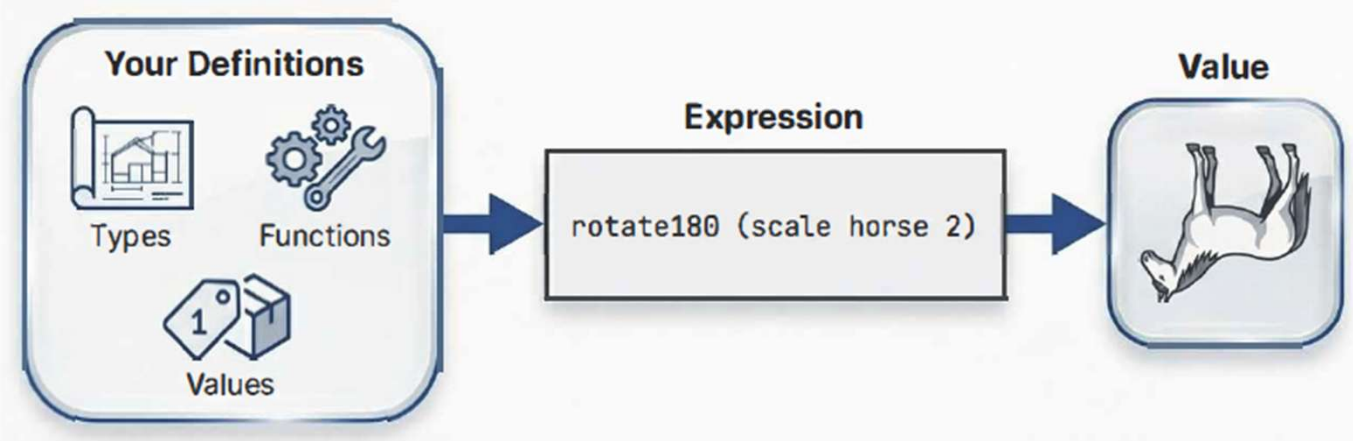
```
2  -- define named constants:
3  r1 = 1
4  r2 = b^2 + 1/b where b = 2
5
6  -- define a function:
7  diff a b = abs (a - b)
8
9  r3 = diff (diff a a) a  where a = 1
10 r4 = diff (diff a b) b  where a = 1; b = 1
11
12 -- lists by enumeration
13 r5 = [1,3,4,1]
14 r6 = [1..5]
15
16 inc n = n + 1
17
18 r7 = map inc [1..3]
19 r8 = map (diff 2) [1..3]
20 r9 = map sqrt [1..3]
21
22 r10 = zip [1..3] (map sqrt [1..3])
23 r11 = zip [1..3] (map sqrt [1..2])
24
25 r12 = print [1..3]
```

Functional PLs – Haskell mini demo (2/2)

```haskell
name = "Alice"
-- "if" has a special syntax but otherwise a typed version of Lisp's "if":
name2 = if name /= "" then name else "no name"

pname3 = print "Bob"

myprogram = print (1 + m)  -- compiler error: m undefined
m = 1                      -- unless this line is also present

-- sequencing several imperative programs:
prg1 = do
  print "hello "
  print name  -- level of indentation is important

-- the same, but using algebra of imperative programs:
prg1' = sequence_ [print "hello ", print name]

-- one imperative program passing value to another:
prg2 = do
  line <- getLine
  putStrLn ("you typed: " ++ line)

main =
  do
  putStrLn name -- like print, but only for strings
  -- putStrLn pname3 -- Couldn't match type ...; Expected type: String; Actual type: IO ()
  pname3 -- in Lisp: eval pname3
  myprogram; prg1; prg2 -- sequencing, like 3 lines
```

The Essence of Haskell Programming

- A Haskell program is a collection of type and function definitions. We use these definitions- including functions, types, and values-to model objects and operations in our problem domain.
- The 'program' runs by evaluating a final expression using this collection of definitions.



The Payoff: Clarity, Safety, and Power

Clarity (No Side-Effects)

Functions only compute a result based on their inputs; they don't change other parts of the system unpredictably. This makes their behavior easy to understand and test.

Safety (Parallelism)

With no shared state for different functions to modify, running code on multiple cores becomes vastly simpler and safer.

Power (Composition)

Small, independent functions are like LEGO bricks—easy to test individually, reuse in new contexts, and refactor without breaking the entire system.

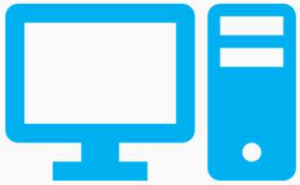
Logical PLs

- Even more abstract model of computation than functional PLs
- A computer is viewed as a semi-intelligent agent which has access to knowledge (in a programme)

Logical PLs

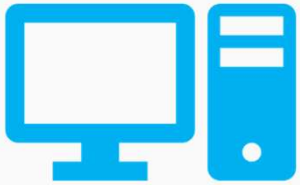
- Even more abstract model of computation than functional PLs
- A computer is viewed as a semi-intelligent agent which has access to knowledge (in a programme)

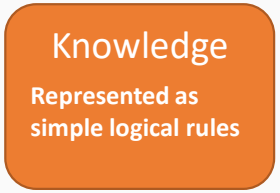
Computer
(Semi-Intelligent agent)



Logical PLs

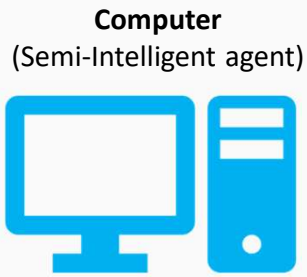
- Even more abstract model of computation than functional PLs
- A computer is viewed as a semi-intelligent agent which has access to knowledge (in a programme)

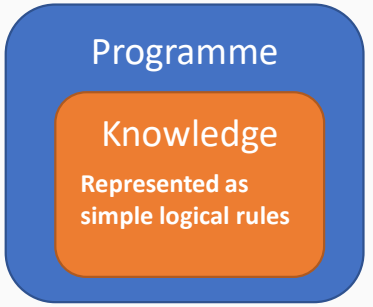




Logical PLs

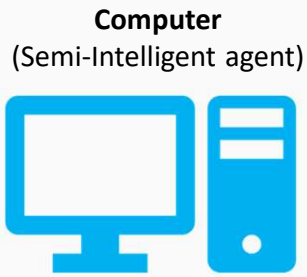
- Even more abstract model of computation than functional PLs
- A computer is viewed as a semi-intelligent agent which has access to knowledge (in a programme)

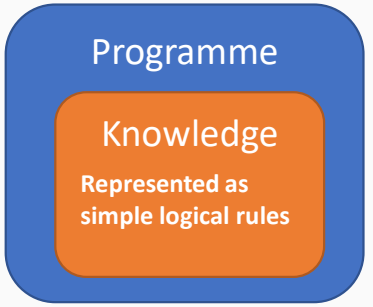




Logical PLs

- Even more abstract model of computation than functional PLs
- A computer is viewed as a semi-intelligent agent which has access to knowledge (in a programme)
- The agent can be asked a question and it will do its best to deduce an answer from the knowledge available in the program





Programme Execution

- Generally two different ways of execution
  1. Interpreter based
  2. Compiler based

Programme Execution

Interpreter based Execution

- Source code translated to machine code line by line
- Performs prompt analysis of time to analyse source code
- Slow code execution
- Debugging is relatively easier due to line-by-line translation
- Examples: Python, Prolog, Java

Compiler based Execution

- Entire source code translated to machine code at same time
- Consumes more time to analyse source code
- Much faster code execution
- Debugging relatively harder since error can be anywhere in program
- Examples: C, C++, Java, Haskell

Source Material

1. Michal Konecny's notes and slides on the same topic
2. Haskell and Functional Programming material is based primarily on Haskell: the Craft of Functional Programming (3rd Edition) by Simon Thompson