FIT2102
Programming Paradigms
Lecture 1

Intro - Levels of Abstraction
Assembler - the machine level
JavaScript and functions

Faculty of Information Technology

MONASH University

FIT 2102 Structure

We will:

- learn to think about programming languages as providing different abstractions of computation and differentiate between *syntax* and *semantics*
- learn about important programming paradigms that provide different models for computation
  - Imperative
  - Functional
  - Declarative
- study several languages that are distinguished by the types of abstractions they provide
- study some theory (lambda calculus / a little bit of category theory) that allow us to think abstractly about programming

Schedule

Week 1:
- Intro
- Assembler - the machine level
- JavaScript and functions

Week 2:
- JavaScript, objects and higher-order functions

Week 3:
- Compiled vs Dynamic Languages
- TypeScript

Week 4:
- Lazy lists, FRP, Assignment 1

Week 5:
- Higher-order Functions
- Combinators / Currying / Lambda Calculus

Week 6:
- Purescript and Haskell

Week 7:
- Assignment 1 due
- Haskell!

Week 8:
- Assignment 2

Week 9:
- Haskell

Week 10:
- Haskell

Week 11:
- Assignment 2 due Friday, October 13th

Week 12 (Guest lecture):
- Constraint Programming - MiniZinc

Why don’t we study language X?

The most important part of this course is the concepts we learn which should be transferable to lots of different languages (*semantics* vs *syntax*).

We can’t study every popular language in 12 weeks. The ones we do look at are chosen because, either:

- They are immensely popular, have interesting features, are used in interesting ways, and you will undoubtedly encounter them in the real world (JavaScript/TypeScript); or,
- They represent interesting examples of their paradigm (PureScript/Haskell/MiniZinc).

Syntax vs Semantics

```python
# python code:
def sumTo(n):
    sum = 0
    for i in range(0,n):
        sum = sum + i
    return sum
```

```javascript
// JavaScript code:
function sumTo(n) {
    let sum = 0;
    for(let i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

What is (and isn't) FIT2102

We **are** going to learn some important concepts that will give you a deeper understanding of:

- what programming is
- how to do it well
- where it comes from
- where it might be going in the future

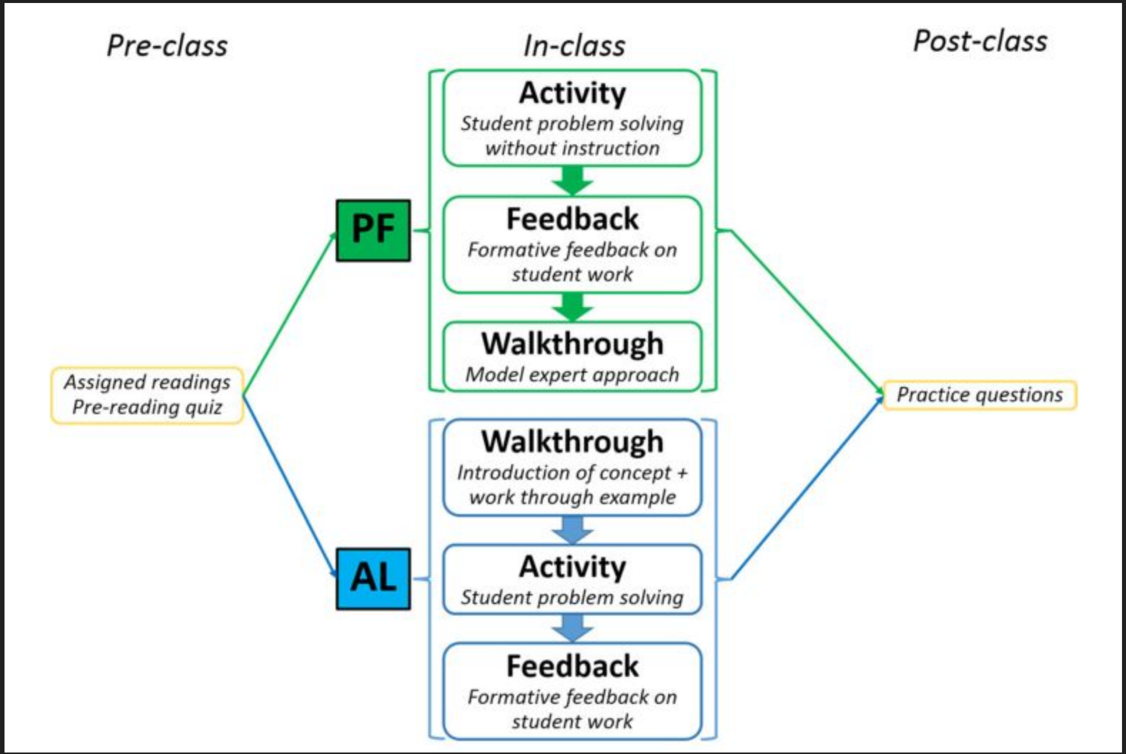
FIT 2102 is **not** a complete "Programming Languages" course.
We are not learning how to:

- make a compiler (though we will discuss how compilers work to some degree)
- design a new language (though we will reflect on language design)

Pedagogy

Productive failure (PF)

Active learning (AL)



Chowrira et al., *“Productive Failure”*, Science of Learning, NPJ, 2019

Assessment

- Lecture and lab activities 20%
    - Lecture assessment based on attendance roll and participation in Moodle forums
    - Previous week's lab exercises and take-home work will be checked by the tutors at the **start** of each lab
- Assignment 1 20%
- Assignment 2 20%
- Exam 40%

Lab Assessment

- Most lab tasks require only a few lines of code
- No one problem should take more than around 30 minutes, most a lot less
- If you find you have not been able to progress on a particular problem after significant time
  - **write down a question in a comment in the code**
  - **try to break the question into sub-questions**
- If still unable to make progress, bring the questions to the lab.
- Your tutor will answer the questions and give you credit for trying (if the question indicates some thought).

Learning Outcomes for Lecture 1

After lecture 1 you should be able to:

- Explain the difference between Syntax and Semantics in programming languages
- Explain the need for abstraction from machine instructions to high-level languages
- Contrast imperative loops with recursive loops
- Explain how the stack is used in functions and how recursive functions can overflow the stack
- Create small JavaScript programs with functions

The Good Old Days

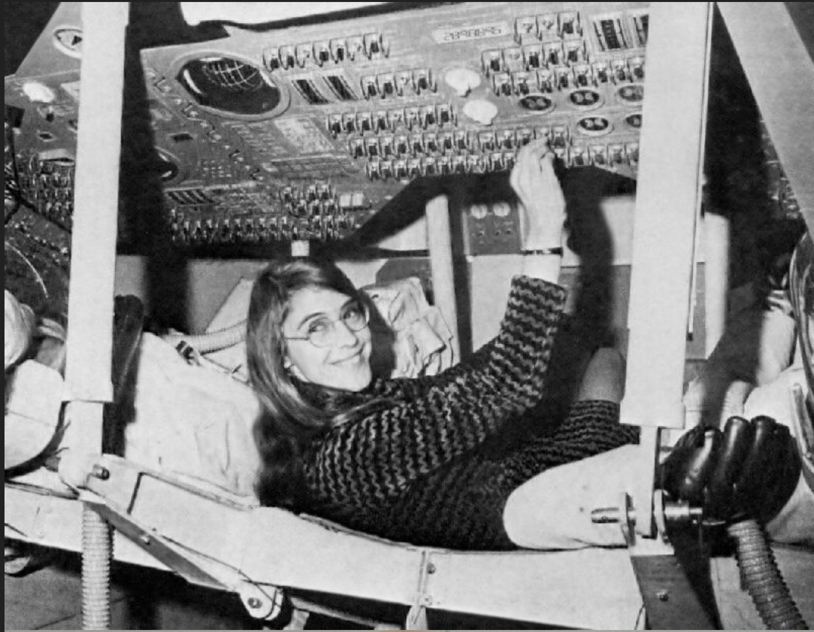
[Margaret Hamilton](#)
[Lead Apollo flight software designer](#)

Engineers built seriously complex software at the Machine level

[Apollo 11 Source Code](#)

```
INDEXI          DEC     4               # ********** DON'T  ***********
                DEC     2               # ********** TOUCH  ***********
                DEC     0               # ********** THESE  ***********
                DEC     4               # ********** CONSTANTS *********
```

[source](#)



Levels of Abstraction

“High-Level” Languages - compiler or interpreter transforms human readable instructions to machine operations

Assembly Language - still requires a compiler, but operations correspond one-to-one with machine operations

Machine Language - operations and their arguments (operands) represented by binary numbers and executed either directly in hardware or by a *microprogram* embedded in the microprocessor

Basic computer architecture

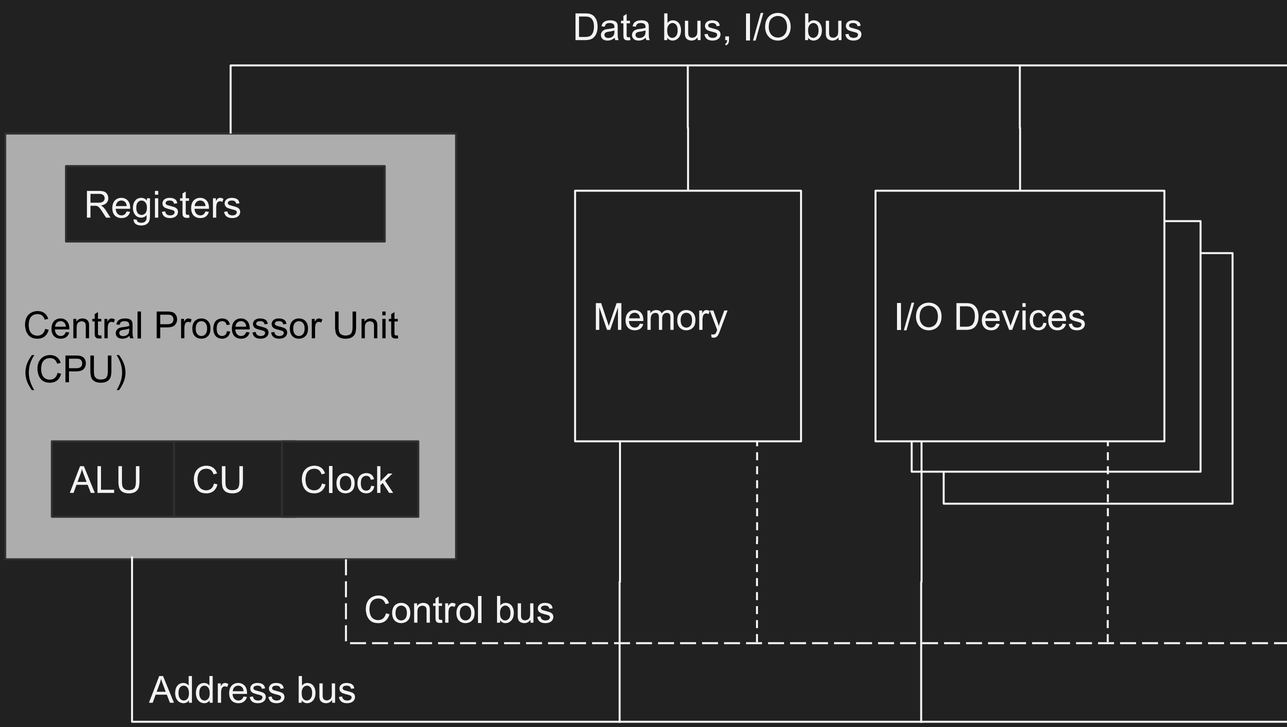
ALU - Arithmetic Logic Unit

CU - Control Unit

Clock - Synchronises CPU operations with other system components

Data bus - transfers instructions and operations between CPU and memory

Address bus - When the CPU needs to read or write to a memory location it specifies that location on the address bus

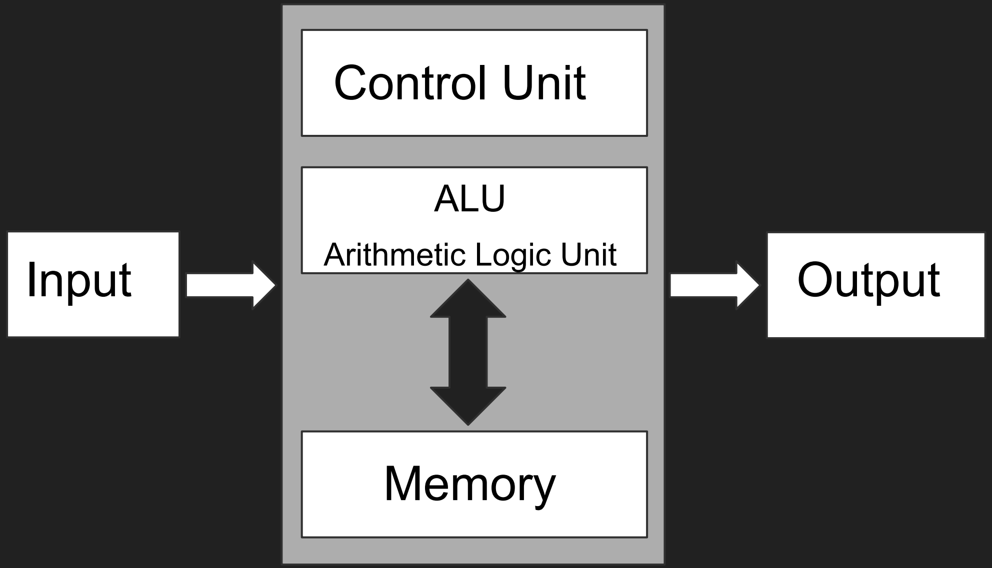


The von Neumann model

A model for computation closely matching actual computer architecture.

Has in common with the Turing Machine model an imperative, “instruction following” paradigm.

We will be learning about an alternative model for computation: the Lambda Calculus



Not Examinable in 2019

Instruction Execution Cycle

1. CPU **fetches instruction** from *instruction queue*, increments instruction pointer
2. CPU **decodes instruction** and decides if it has operands
3. If necessary, CPU **fetches operands** from registers or memory
4. CPU **executes the instruction**
5. If necessary, CPU stores result, sets status flags, etc.

x86 Data Registers

Not Examinable in 2019

Just some of the registers on a modern x86 chip...

Name	64-Bit	32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
Accumulator	RAX	EAX	AX	AH	AL
Base	RBX	EBX	BX	BH	BL
Counter	RCX	ECX	CX	CH	CL
Data	RDX	EDX	DX	DH	DL

Not Examinable in 2019

Let’s make a program!

```
.386                                 ; choose 32 bit mode
.model flat, stdcall                 ; execution model (don't worry about it for now)
ExitProcess PROTO, dwExitCode:DWORD ; declares this special external procedure and the type of its parameter

.code                                ; opens the code section
main PROC                            ; here's where our "main" procedure begins
      mov eax, 5                     ; move 5 into eax register
      add eax, 6                     ; add 6 to the value in the eax register
      invoke ExitProcess, eax        ; exit returning eax as the result code
main ENDP

END main
```

```
The program '[3268] AssemblerProject.exe' has exited with code 11 (0xb).
```

result

Not Examinable in 2019

Basic instructions: arithmetic

```
[label:] mnemonic [operands] [; comment]
```

Operands can be memory locations, registers or numeric literals

Mnemonic	Operands	Description
mov	dest, src	Move (copy) contents of src into dest
add	y, x	Add contents of x to y (result in y)
sub	y, x	Subtract x from y (result in y)
mul	x	Multiply eax by x (result in eax)
div	x	Divide eax by x, result in eax, remainder in edx
xor	y, x	Bitwise exclusive or (Tip, to zero a register: xor eax, eax)

Not Examinable in 2019

Basic instructions: jumps

Mnemonic	Operands	Description
jmp	label	Jump unconditionally to label
loop	label	If ecx > 0 jump to label and decrement ecx
cmp	x, y	Compare x and y and set the flags register accordingly
je	label	Jump to label if result of previous cmp operation was equal
j[ne|l|le|g|...]	label	Jump to label if ↑ not equal, <, <=, >, etc...

Not Examinable in 2019

Basic instructions: stack operations

Code		Address	Stored values
`push 3`	esp →	00001000	00000003

Mnemonic	Operands	Description
push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

Not Examinable in 2019

Basic instructions: stack operations

Code

```
mov eax,5
push eax
```

	Address	Stored values
esp →	00000FFC	00000005
	00001000	00000003

Mnemonic	Operands	Description
push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

Not Examinable in 2019

Basic instructions: stack operations

Code		Address	Stored values
`mov var1,1`	esp →	00000FF8	00000001
`push var1`		00000FFC	00000005
		00001000	00000003

Mnemonic	**Operands**	**Description**
push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

Not Examinable in 2019

Basic instructions: stack operations

Code		Address	Stored values	Result
`pop var2`	esp →	00000FFC	00000005	eax = 1
		00001000	00000003	

Mnemonic	Operands	Description
push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

Not Examinable in 2019

Basic instructions: stack operations

Code	Address	Stored values	Result
`pop eax`	esp → 00001000	00000003	var2 = 5

Mnemonic	Operands	Description
push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

Not Examinable in 2019

Basic instructions: procedures

```
.386
.model flat, stdcall
ExitProcess PROTO, dwExitCode:DWORD

.data
      varA DWORD 1
      varB DWORD 2
      varC DWORD 3
.code
main PROC
      mov ebx, varA
      mov ecx, varB
      mov eax, varC
      call sum3                         ; call the procedure
      invoke ExitProcess, eax           ; result is 1+2+3=6
main ENDP

sum3 PROC
      add eax, ebx
      add eax, ecx
      ret                               ; need to end procedures (other than main) in ret
sum3 ENDP

END main
```

JavaScript

JavaScript (aka EcmaScript), is an incredibly ubiquitous language that has supported “functions as values” since its inception.

As a result, server and client side JavaScript in the wild makes heavy use of Functional Programming (FP) patterns.

You will see sophisticated FP influence in many JS libraries
(JQuery, React, D3, RxJS, Mocha and *many* more).

FP makes these libraries ***flexible*** and ***robust***.

JavaScript 101: defining variables and functions

```
const z = 1;  // constant (immutable variable) at global scope
/**
* define a function called "myFunction" with two parameters, x and y
* which does some silly math, prints the value and returns the result
*/
function myFunction(x, y) {
   let t = x + y; // t is mutable
   t += z;  // += adds the result of the expression on the right to the value of t
   const result = t // semi colons are not essential (but can help to catch errors)
   console.log("hello world") // prints to the console
   return result; // returns the result to the caller
}
```

JavaScript 101: if statements

```
/**
* get the greater of x and y
*/
function maxVal(x, y) {
   if (x >= y) {
        return x;
   } else {
        return y;
   }
}
```

JavaScript 101: if statements

```
/**
* get the greater of x and y
*/
function maxVal(x, y) {
    if (x >= y) return x;
    return y;
}
```

JavaScript 101: if statements

conditional expression syntax:

```javascript
/**
* get the greater of x and y
*/
function maxVal(x, y) {
    return x >= y ? x : y;
}
```

JavaScript 101: Loops

```
/**
* sum the numbers up to and including n
*/
function sumTo(n) {
   let sum = 0;
   while (n) {        // because Boolean(0) === false
       sum += n--;    // operator fun!  Cheatsheet coming...
   }
   return sum;
}
```

JavaScript 101: Loops

```javascript
/**
* sum the numbers up to and including n
*/
function sumTo(n) {
   let sum = 0;
   for (let i = 1; i <= n; i++) {
       sum += i;
   }
   return sum;
}
```

JavaScript 101: Basic Operator Cheat Sheet

Binary Operators:

```
x % y   // modulo
x == y  // loose* equality
x != y  // loose* inequality
x === y // strict* equality
x !== y // strict* inequality

a && b   // logical and
a || b   // logical or

a & b   // bitwise and
a | b   // bitwise or
```

Unary Operators:

```
i++      // post-increment
++i      // pre-increment
i--      // post-increment
--i      // pre-increment
!x       // not x
```

In-place math operators:

```
x += <expr>
// add result of expr to x
// also -=, *=, /=, |=, &=.
```

Ternary Conditional Operator:

```
<condition> ? <true result> : <false result>
```

```
* Loose (in)equality means type conversion may occur
  Use strict (in)equality if type is expected to be same
```

Live coding exercise 1:

To be announced...

JavaScript 101

Recursive Loops:

```
/**
* sum the numbers up to n
*/
function sumTo(n) {
   if (n === 0) return 0;   // base case
   return n + sumTo(n-1);   // inductive step
}
```

We consider this recursive loop a more "declarative" coding style than the imperative loops.

It is closer to the ***inductive definition*** of sum than a series of steps for how to compute it.

- ***No mutable variables*** used
- Each expression in this code is "***pure***": it has no ***effects*** outside the expression.
- Therefore: this code has the property of ***referential transparency***.
- The code succinctly states the ***loop invariant.***

A rather serious caveat:
Too many levels of recursion will cause a ***stack overflow***.

JavaScript 101

Recursive Loops:

```
/**
* sum the numbers up to n
*/
function sumTo(n) {
   return n ? n + sumTo(n-1) : 0;
}
```

We consider this recursive loop a more "declarative" coding style than the imperative loops.

It is closer to the ***inductive definition*** of sum than a series of steps for how to compute it.

- ***No mutable variables*** used
- Each expression in this code is "***pure***": it has no ***effects*** outside the expression.
- Therefore: this code has the property of ***referential transparency***.
- The code succinctly states the ***loop invariant.***

A rather serious caveat:
Too many levels of recursion will cause a ***stack overflow***.

Live coding exercise 2:

To be announced...

JavaScript 101: First higher-order function

```
function square(x) {
   return x * x;
}

function sumTo(n, f) {
   return n ? f(n) + sumTo(n-1, f) : 0;
}


sumTo(10, square)
```

> 385

Conclusion

Assembly language offers minimal abstraction over the underlying computer architecture, it offers:

- the ability to name operations and memory locations symbolically;
- the ability to define procedures (more recently);
- conveniences for dealing with arrays and macros (not examined here)

C is adds more understandable syntax,
but is still close to the machine execution model.

Languages we examine in future lectures will depart further and further from the underlying (von Neumann) computer architecture.

Conclusion (cont...)

JavaScript is basically an imperative language with C or Java-like syntax, except that it is:

- Interpreted
- Functions are objects which can be assigned to variables

In coming weeks we will incorporate more "functional" abstractions into our JavaScript coding, before moving onto a language which significantly departs from the imperative model.

Imperative programming involves telling the computer how to compute step-by-step

Declarative programming describes what you want to do, not how you want to do it