MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Programming Languages (Part 1)

Daniel Playne
d.p.playne@massey.ac.nz

What is a Programming Language?

A vehicle for the user to tell the computer what to do to solve the user's problem.

Enables the user to bridge the gap between the ***problem domain*** and the ***machine domain***.

Problem domain is all about the application.
Machine domain is about the memory, CPU, peripherals and the computer's capabilities.

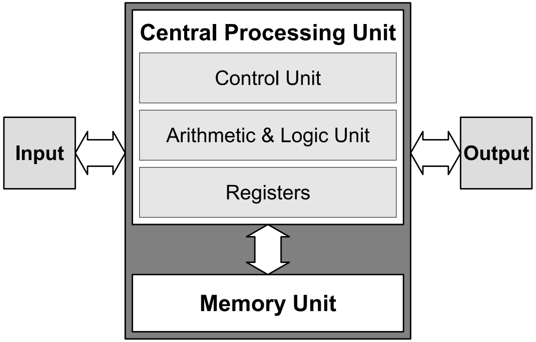
Who is the Programming Language for?

The ***human*** - a language should be readable and both ***memorable*** and ***understandable*** for both the human who wrote it and the other humans who inevitably have to read it.

The ***computer*** - just needs a compiler that understands the particular language and can compile it into machine code. Sometimes this does not even require a compiler.

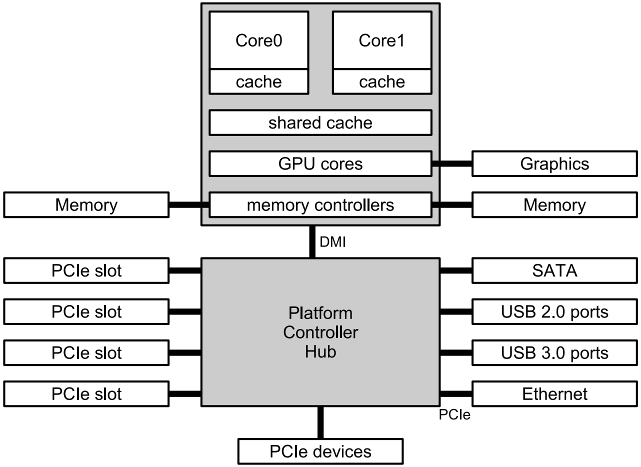
What is a Computer?

You may see computers described with some simplified architecture such as the von Neumann architecture (named after John von Neumann).



Real Architectures

Actual hardware machines usually look rather different.



Models of Computation

In fact, programmers usually think at an even more abstract level and reason in terms of a ***model of computation*** rather than an actual machine.

The model of computation you are probably most familiar with is the ***Random Access Machine***:

- ***Control Unit*** - holds a list of statements (the program).
- ***Program Counter*** - determines the next statement to be executed.
- ***Memory*** - an infinite sequence of registers.

Models of Computation

The ***Random Access Machine*** is closely related to the design of the physical machines we use but it is certainly not the only valid model of computation.

Different programming languages are greatly influenced by the model they are based on.

We will also see that sometimes the details of which type of machine is running our program does matter.

Turing Machines

Another *model of computation* was described by Alan Turing and is called ***Turing Machines*** (although he always just called them machines).

These machines essentially describe modern computers - approximately ten years before physical computers where ever actually built (most probably with Turing's input).

Turing Machines

There are a number of slightly different versions or definitions of what a Turing Machine is.

Essentially a Turing Machine consists of a ***head*** that moves backwards and forward across a ***tape***.

At each stage, the head will be in a certain ***state*** and will select a certain action from the ***instruction table*** based on the state of the head and the ***symbol*** written on the tape.

Turing Machines

Tape - the tape of a Turing machine is considered to be an infinitely long tape divided into sections, each of which has a symbol written on them or are empty. The ***head*** can read/write a symbol off/onto each section of the tape.

Alphabet - each symbol written on the tape is part of some finite alphabet. There are no particular restrictions on this alphabet but it must be finite and contain at least two symbols - one of which is the blank symbol.

Turing Machines

Head - the head of a Turing Machine is always positioned on a particular section of the tape and can read from the tape or write to the tape at that section. The head can also move left or right on the tape.

States - a Turing Machine also has a finite set of states that the machine can be in. One of these states is designated to be the starting state of the machine. The state can be changed as the machine steps forward based on the instruction set. Most Turing machines also have a Halt state which signifies that the machine has stopped.

Turing Machines

At each step of computation, a Turing Machine will perform the following actions:

- ***read*** the symbol on the tape at the current position of the head.
- ***write*** a symbol onto the tape (at the current position).
- ***move*** the head left or right along the tape by one section.
- ***change*** the state of the head.

This process is then repeated until the machine ***halts***.

Turing Machines

Turing machines were not intended as an actual computer design, but a way to formally describe algorithms.

But naturally people have since made them:

`https://www.youtube.com/watch?v=E3keLeMwfHY`

Turing Machines

There are fairly limited options for creating programming languages for Turing machines as programs mostly just consist of sets of instructions.

Each instruction considers a current state and system and maps them to a new state, symbol and a movement.

```
current_state; current_symbol; new_state; new_symbol; move
current_state; current_symbol; new_state; new_symbol; move
...
```

Turing Machines

For example, consider the following program for a Turing Machine ($\wedge$ represents the blank symbol and +1 represents a move right).

```
a;  ∧;  b;  0;  +1
b;  ∧;  c;  ∧   +1
c;  ∧;  d;  1;  +1
d;  ∧;  a;  ∧   +1
```

The ***random-access machine*** is in many ways an extension of Turing machines and thus imperative programming language could be considered as indirectly based on TMs.

Lambda Calculus

Another *model of computation* (developed at the same time as Turing machines) which has lead to a class of languages is Lambda Calculus.

λ-calculus is a simple but powerful system based on function abstraction and application.

Function abstraction generalises expressions through the introduction of names.

Function application evaluates generalised expressions by giving names particular values

Lambda Calculus

λ-calculus programs consists of function abstractions and applications and usually end up with expressions that look like this:

```
((λfunc.λarg.(func arg) λx.x) λs.(s s))
```

This model of computation has led to the development of functional programming languages which lead to many programming concepts that some imperative programmers find unintuitive.

Classifying Programming Languages

When studying and discussing programming languages, it is often useful to try to classify languages and group them together based on some common features etc.

Programming Paradigms are one way of classifying a programming language.

The term has become somewhat abused but in Computer Science is usually - a coherent set of methods that have been found to be (more or less) effective in handling a particular type of problem.

Programming Paradigms

Usually a paradigm can be described by a simple core guiding principle or a model of computation that paradigm is based on.

Be aware that not all groupings or models that are declared to be paradigms are equally coherent.

Paradigms often only become apparent *post hoc* once the fuss has died down.

Programming Paradigms

The are four commonly accepted programming paradigms:

- Imperative Paradigm
- Object-Oriented Paradigm
- Functional Paradigm
- Logic Paradigm

Be aware that others do exist and are constantly being proposed and argued for/against.

Imperative Paradigm

The ***imperative*** programming paradigm is based on ***statements*** that change the ***state*** of a program (***Random Access Machine***).

The imperative programming paradigm is still very much influenced by the workings of a machine and introduces higher-level concepts over assembly code.

A major difference is that each statement doesn't necessarily map to a single machine instruction.

Machine Code

Machine code is about as low level a language as we can get. Just a list of numbers.

Specifies hardware machine instructions such as:

- Move the value in Register X to memory location Y
- Add two registers together leaving the answer in another
- Branch two instructions forward if register X is zero

Often hexadecimal (base 16) translates into values that control the hardware.
Luckily we rarely have to work at this level.

Assembly Language

A set of textual macros and mnemonics for the numbers used in machine code.

Often manufacturers supply quite elaborate tools for developing in assembly for their architecture.

Often only needed at the lowest bootstrap level when developing a new computer.

Maybe used to optimise the speed of some very important code - e.g. a device driver.

Example

```
Machine (Hex):
23fc 0000 0001 0000 0040

0cb9 0000 000a 0000 0040
6e0c
06b9 0000 0001 0000 0040
60e8
```

```
Assembly:
  movl   #0x1, n
compare:
  cmpl   #0xa, n
  bgt    end_of_loop
  addl   #0x1, n
  bra    compare
end_of_loop:
```

High-Level Languages

More readable than assembler and gets the compiler to do the loop address calculations for us.

FORTRAN:

```
  DO 2 N = 1,10
2 CONTINUE
```

COBOL:

```
    MOVE 1 TO N.
AGAIN.
    IF N IS EQUAL TO 10
       GOTO END-DEMO.
    ADD 1 TO N.
    GOTO AGAIN.
END-DEMO
```

High-Level Languages

A loop that does nothing except change the loop counter. There may be a **NOP** or no-operation instruction.

```
C:
for(n = 1; n <= 10; n++) {

}
```

```
Ada:
for N in 1..10 loop
   null;
end loop;
```

High-Level Languages

All programming languages are attempts to map a problem to the machine.

Links the user's problem specific language to the workings of the machine.

Some early high-level languages - Fortran, Cobol, C use ideas (and syntax) that still exist in modern languages and build on assembler ideas.

Object-Oriented Paradigm

Object-Oriented programming was heavily influenced by SIMULA67 and Smalltalk80. Object-oriented languages are based more generally on a model of interacting objects.

Object-oriented languages are strongly based on the concepts of objects which may contain data and methods.

A language that is object-oriented generally provides support for the following key language features: abstract data types, inheritance and dynamic binding of method calls.

Functional Paradigm

The ***functional*** paradigm is based on ***lambda calculus*** and is the basis of many non-imperative languages and language features.

One of the fundamental characteristics of an imperative programs is that they have state, which changes during the execution of the program. To understand what an imperative program will do, the programmer must understand how the state will change.

Functional programs are argued to be easier to understand (and more likely to be correct) because their meaning is independent of their context.

Logic Paradigm

The ***logic*** paradigm is based on ***first-order logic***. Propositions are defined as either ***atomic*** propositions or ***compound terms***.

Programming in a logic language never defines ***how*** a result is to be computed. Instead they describe the form of the result and assume that the computer can somehow work out how to compute the result.

Requires logic languages to concisely provide the necessary information and method of inference.

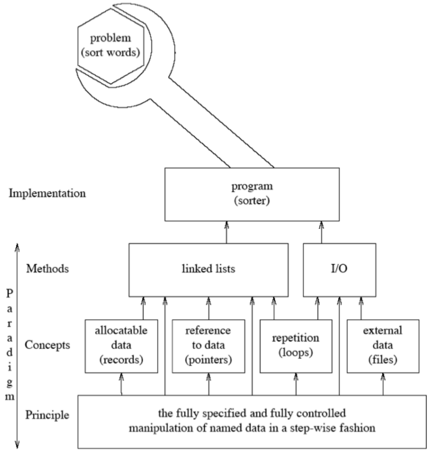
Declarative

The Functional and Logic programming paradigms are sometimes grouped together and referred to as ***Declarative languages***

Declarative languages emphasize ***what*** is to be done rather than ***how*** it is to be done.

Languages that focus ***how*** are called ***operational*** or ***non-declarative***. The imperative (procedural) and object-oriented languages tend towards the operational.

Programming Paradigms


problem (sort words)
Implementation
program (sorter)
Methods
linked lists
I/O
Paradigm
Concepts
allocatable data (records)
reference to data (pointers)
repetition (loops)
external data (files)
Principle
the fully specified and fully controlled manipulation of named data in a step-wise fashion

From: Bal & Grune, Programming Language Essentials, Addison-Wesley

Compilers

A ***compiler*** is a tool (program) that can convert a programming written in a high-level language into machine code (or other suitable representation).

Turns our text (that we understand) into numbers (that the machine uses).

More on this in the next lecture.

Summary

- Models of Computation
    - Random Access Machine
    - Turing Machines
    - Lambda Calculus
- Programming Paradigms
    - Imperative
    - Object-Oriented
    - Functional
    - Logic