Theory of programming languages

- Programming languages provide us with a way of expressing solutions to problems
- The nature and features of the language shape the kinds of solutions we can express, and how easily
- Learning and using very different styles of language helps us spot solutions to problems using vastly different approaches
- Understanding a problem and a variety of languages lets us assess which languages are most suitable for specific problems
- Even if our current language doesn’t directly implement features of some other language, knowledge of those features might still allow us to build a solution using the same styles/techniques

Programming paradigms

- Programming paradigms: different styles/approaches to programming
- We tend to associate specific languages with specific paradigms, but most languages support multiple paradigms
- Ask 5 academics and you'll get 7 answers on what the core paradigms are – we'll discuss four in some detail
- Well group the paradigms into imperative and declarative
- Imperative: solutions exlplicitly control the specific order/sequence of instructions/steps to be taken
- Declarative: describe operations and the conditions under which they are run, specific sequencing not provided

Imperative paradigms

- Procedural: describe step-by-step the exact sequence of actions to perform, program state explicitly controlled through assignment of values to variables and memory
- Pure procedural: assembly language, C, bash, etc
- Object oriented: encapsulate operations and data associated with objects, typically the operations being expressed in a procedural form
- Pure OO: everything is always an object (e.g. smalltalk)
- Hybrids: most “OO” languages (C++, Java, C#, etc etc)

Declarative paradigms:

- Logic programming: describe the universe by set of facts and rules, then make queries about the universe, which logic engine tries to answer by combining facts/rules
- Logic programming languages: prolog
- Functional programming: everything is either a function call or data, programs consist of compositions of functions, no explicit use of stored state (variables), no side effects (no pass-by-reference)
- Pure functional languages: haskell
- Hybrid languages: common lisp, scheme, erlang, etc

Language implementation

- The actual implementation of a language has a huge impact on how effectively the language can be used in different situations
- Impacts runtime behaviour/limitations, speed, memory use, reliability, security, ease/difficulty of tool development (debuggers, profilers, compilers, interpretters, etc)
- Developers need to be aware of implementation decisions made for their language, platform, compiler version so they are aware of the implications
- Knowing how features can be implemented also lets dev mimic a feature in languages that don’t otherwise support it

Explore programming languages

- Incredible diversity of programming languages and styles out there
- Huge library of programming languages (802) many with code examples for each of many (1070) different tasks:
- rosettacode.org/wiki/Category:Programming_Languages
- rosettacode.org/wiki/Category:Programming_Tasks

C example, stack/push

```
#include <stdlib.h>
struct Node { // define nodes for our stack
   int data;
   struct Node* next;
};
struct Node* push(struct Node* S, int d) { // push function
   struct Node*n;
   n = (struct Node*)malloc(sizeof(struct Node));
   If (!n) return S;
   n->data = d;
   n->next = S;
   return n;
}
// define an empty stack then call push and update the stack
struct Node *mystack = NULL;
mystack = push(mystack, 10);
```

C++ STL example, stack/push

```
#include <stack>
...
// use the STL to create a stack of ints
stack<int> mystack;
// use the stack's push method to push 10
mystack.push(10);
```

Lisp example, stack/push

```
; define a push onto an existing stack
(defun push (S i) (cons i S))
; define a new stack creationg
(defun newStack ( ) '())
; push 10 onto a newly created stack
(push (newstack) 10)
```

Prolog example, stack/push

```
stack([]).  % fact for an empty stack
stack([_|S]) :- stack(S). % fact for non-empty stack
push(E, [E|S], S) :- stack(S). % rule for a push

% issue query to push 10 on a new empty stack
stack(S), push(10, Result, S).
```