COMP2212 Programming language concepts

Lecture 1 - Introduction
Assessment

- **Coursework** - programming language design
  - Worth 40% of module mark
  - Spec released in two weeks
  - Done in pairs, find a partner now
  - You will need to write Haskell code
- **Exam** - worth 60% of module mark
Reasons for studying PL concepts

❖ increased capacity to express ideas
  ❖ different concepts of a programming language will cause you to solve problems in different ways
  ❖ on the other hand, people find it more difficult to conceptualise structures for which there is no explicit language support

❖ be able to choose the right language for the job
  ❖ different languages have different strengths and weaknesses
Reasons for studying PL Concepts

❖ increased ability to learn new languages
❖ technology continues to change
❖ knowing a wide range of languages will make it easier to master languages of the future
❖ better use of already known languages
❖ understand parts of language that were mysterious
❖ use techniques from other language in another
Application domains

- **Scientific computing**
  - Fortran historically used for mathematical applications
  - Mathematica, MatLab
  - modern functional languages: OCAML, Haskell

- **Business applications**
  - COBOL is still used…
  - more recently, languages for business processes: BPEL, web services, etc.
Application Domains

- **Artificial Intelligence**
  - LISP (a functional language)
  - logic programming (e.g. Prolog)

- **Systems**
  - C/C++

- **Web**
  - markup languages: HTML, XHTML
  - scripting languages: Perl, PHP, Ruby, Javascript, …
Major programming language families

- Imperative programming languages
  - C, C++, Java, C#, Pascal, ….

- Functional programming languages
  - OCaml, Haskell, ML, Lisp, Scheme, F#, …

- Logic/Declarative programming languages
  - Prolog and variants
Mergesort in pascal

const INFTY = maxint;
var a, b, c : array [1..10000] of integer;

procedure merge (l, m, r : integer);
var i, j, k : integer;
beginn
  for i := l to m do b[i] := a[i];
i := l; b[m+1] := INFTY;
  for j := m+1 to r do c[j] := a[j];
j := m+1; c[r+1] := INFTY; k := l;
  while (b[i]<INFTY) or (c[j]<INFTY) do
    if b[i]<c[j] then begin a[k] := b[i]; inc (i); inc(k); end
    else begin a[k] := c[j]; inc (j); inc(k); end;
end;

procedure mergesort (l, r : integer);
var m : integer;
beginn
  if l < r then begin
    m := (l+r) div 2;
    mergesort (l, m);
    mergesort (m+1, r);
    merge (l, m, r);
  end;
end;
open Future;;
open List;;

(* sort a list of integers l with parallelism *)
let rec mergesort (l:int list) : int list =
  (* merge two sorted lists recursively *)
  let rec merge (l1,l2) =
    match (l1, l2) with
    ([], []) -> []
    | (l, []) -> l
    | ([], l) -> l
    | (hd1::tl1, hd2::tl2) ->
      if (hd1 < hd2) then
        hd1::merge (tl1,l2)
      else
        hd2::merge (l1,tl2) in
  (* split a list l in half *)
  let split l : int list * int list =
    match l with
    [] -> ([], [])
    | hd::[] -> (l, [])
    | _ -> let (l1, l2, n) = List.fold_left
      (fun (l1, l2, n) nxt ->
        if (n > 0) then (nxt::l1, l2, n-1)
        else (l1, nxt::l2, n-1)) ([],[],(List.length l)/2) l in
      (List.rev l1, List.rev l2) in
  (* perform mergesort with multiple threads *)
  match l with
  [] -> []
  | hd::[] -> [hd]
  | _ -> let l1, l2 = split l in
    let f = Future.future mergesort l1 in
    let l2' = mergesort l2 in
    let l1' = Future.force f in
    merge (l1', l2')
    ;;
Mergesort in Prolog

mergesort([],[]).  /* covers special case */
mergesort([A],[]).
mergesort([A,B|R],S) :-
    split([A,B|R],L1,L2),
    mergesort(L1,S1),
    mergesort(L2,S2),
    merge(S1,S2,S).

split([],[],[]).
split([A],[],[]).
split([A,B|R],[A|Ra],[B|Rb]) :-
    split(R,Ra,Rb).

merge(A,[]A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :- A =< B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :- A > B, merge([A|Ra],Rb,M).
Software design methodologies

❖ Data oriented

❖ ADTs

❖ object-oriented design: encapsulates data together with code, concepts of class, object, inheritance, ...

❖ Procedure oriented

❖ emphasises decomposing code into logically independent actions, often emphasises in concurrent programming
Cross-fertilisation

- Functional programming concepts are used in imperative languages, e.g. Java closures, C#, Rust
- Features such as monads allow functional programmers to write code in imperative style
- Declarative code (e.g. regexp, SQL, HTML, ...) is sometimes allowed to be intermingled with ordinary code
Evaluation criteria

❖ A language ought to be easy to write programs in, result in readable code, help the programmer to avoid bugs, provide an appropriate level of abstraction, make the code run fast, ...

❖ Readability

❖ how easy is it to read and understand code?
❖ more time is spent on debugging someone else’s code than writing your own
❖ readability vs convenience

❖ obfuscation

```c
int composeAndApply(int (*x)(int), int (*y)(int), int arg) {
    return ((y)((x)(arg)));
}
```
Readability

- Simplicity is in the eye of the beholder
  - depressing fact: most people are intellectually lazy/guarded and actively fight against learning new things
  - familiar things will seem “simpler” even when they are more convoluted
- Feature multiplicity
  - incrementing a variable
- Orthogonality
  - features that are independent of each other
  - e.g. arrays cannot be returned by functions in C, confusion between pointer and array types, etc.
Abstraction

• programming has progressively become more high-level
  • Dijkstra’s “GOTO considered harmful”
  • functions, modules
  • classes, objects
• more abstraction means further separation from machine-level thinking
  • 1980s games programmers understood every single feature of their target hardware
  • no programmer understands the complexities of modern hardware today
    • pipelining, memory models, multi-core, …
• we must rely on smart compilers — and these are difficult to write
Efficiency

- Sometimes certain language features are difficult to implement efficiently, or make it difficult to profile code
  - *e.g.* because of lazy evaluation in Haskell it is sometimes difficult to tell where in your code is the bottleneck
- A good compiler will often do a better job or optimizing your code than you could hope to
- Languages such as C have been around forever and have brilliant *optimising* compilers
Extremely short history of programming languages

- early 1950s: Introduction of Fortran, first widely used high-level compiled programming language
  - IBM 704 hardware with floating-point support
  - extremely primitive type system
- late 1950s: John McCarthy at MIT and LISP - the first functional programming language
  - dominated in AI applications
  - Scheme, a LIPS dialect, is still popular
- 1960s: ALGOL designed through enormous committee effort. Let to BNF, procedural design, orthogonal design of language features (ALGOL68).
  - heavily influenced the design Pascal, Ada, C, Simula, Java, etc.
Extremely short history of programming languages

- 1960s: COBOL for business applications. Emphasises data processing as opposed to control flow.
- 1970s: C and Unix machines
- 1970-80s: BASIC for microcomputers
- 1970s: PROLOG - a declarative approach
- 1970s: Ada, emphasis on security and reliability, exception handling
- 1980s: Robin Milner and ML, everything is typed at compile time via a revolutionary type system
- 1980s: Smalltalk and the discovery of Object-Oriented programming
- 1990s: Java and the JVM - flexibility and portability
- 2000s: Scripting and server-side programming - Perl, Python, PHP, JavaScript, Ruby, …
What’s coming in the future?

- Google is pushing big-data computing, e.g. MAP / REDUCE, with languages designed to support such features (e.g. Go)
- Component-based programming - bridging the gap between software and hardware design
- Importance of libraries (e.g. java.util.concurrent)
- Concurrency is still a challenge, can we do concurrency declaratively, in a way that’s compatible with other programming language features?