

F28PL1 Programming Languages

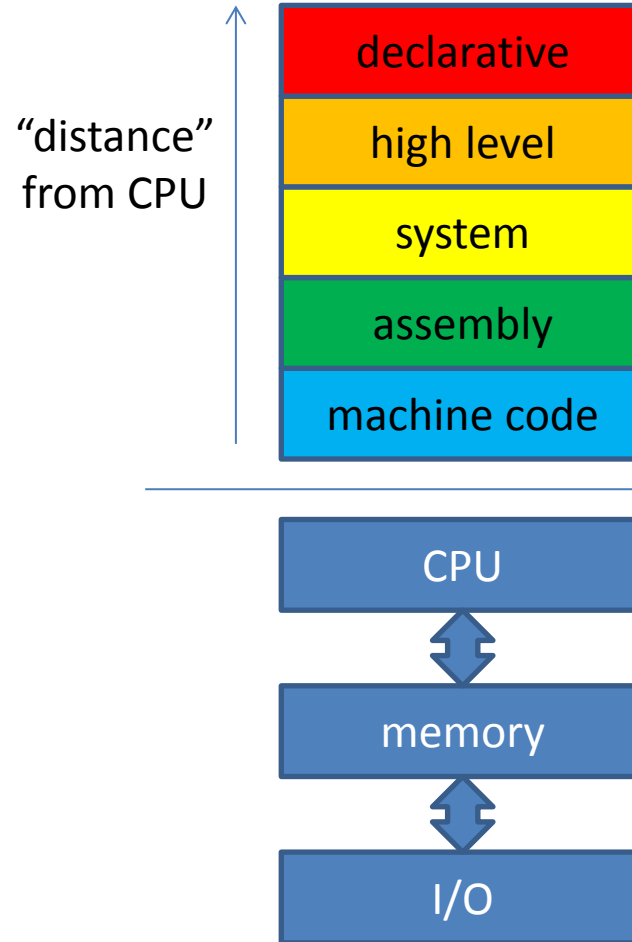
C & Assembly Language Summary

Overview

- started by characterising a computer as a memory machine
- considered programming languages as abstractions from:
 - memory as address/byte associations
 - instructions as sequences of changes to address/byte associations

Overview

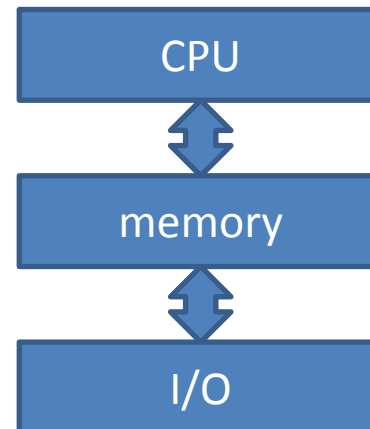
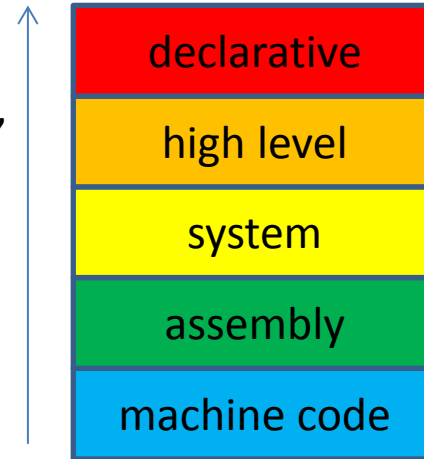
- layers of abstraction
- hierarchy of expressiveness
- tension between:
 - expressive power/ease of use
 - ease of implementation/performance



Overview

- greater distance from CPU →
 - increased expressiveness/succinctness
 - increased implementation effort
 - decreased performance
 - more memory
 - less speed

“distance”
from CPU



Summarising language characteristics

- how does a language abstract away from underlying byte machines?
 - types
 - data abstraction
 - control abstraction
- what are pragmatic consequences of language characteristics?
 - i.e. how do characteristics affect use?

Types

- values & operations
- what are *base* types?
 - e.g. Java: int, float, char etc
- what are *structured* types?
 - e.g. Java: object, array, String etc

Types

- how do types constrain language?
- *weak v strong* typing
 - i.e. whether type associated with entity can change (weak) or can't change (strong)
 - e.g. Java: strong typing
- *static v dynamic* typing
 - i.e. whether types checked at *compile time* (static) or *run time* (dynamic)
 - e.g. Java: static typing

Polymorphism

- type abstraction
 - can types be generalised?
 - *polymorphism* == many shapes
- *ad-hoc* v *parametric* polymorphism
 - ad-hoc == “for this”
 - i.e. language/context specific
 - parametric == controlled by parameters

Polymorphism

- e.g. Java
 - ad-hoc polymorphism
 - operator overloading
 - i.e. can use some operators with different types
e.g. arithmetic
 - parametric polymorphism
 - i.e. generic types with type variables

C summary: types

- strong, static types
 - but... can override types with casts & address manipulation
- base types
 - int, short, long, char, float, double
- structured types
 - array, struct, pointer, union, function
- ad-hoc polymorphism
 - operator overloading & cast

Assembler summary: types

- no types
 - everything is a byte
- *representations* for:
 - numbers, characters etc
- no type checking
 - representation are *conventional*
 - can apply any operation to any byte sequence regardless of representation
 - i.e. ad-hoc polymorphism

Data abstraction

- memory abstraction
- variable as name/value abstraction from address/contents
 - e.g. Java: variables
- where may variables be introduced?
 - e.g. Java: class fields, method formal parameters, block/method bodies, iteration control

Data abstraction

- how may variables be bound to values?
- e.g Java:
 - initialising declaration
 - assignment
 - parameter passing

Data abstraction

- *scope*
 - i.e. where is something visible?
 - *lexical v dynamic scope*
 - i.e. constrained or not by site of definition/declaration
- *extent*
 - i.e. how long does something exist?

Data abstraction

- e.g. Java: lexical scope
 - class field/method names visible via object
 - variables in block/ method body visible in block unless redefined
 - method formal parameter visible in method body only
- e.g. Java: block extent
 - variables only exist in defining block/method body

C summary: data abstraction

- variable
 - name/value association
 - abstracts address/contents
- can still expose low level memory:
 - `&` and `*`
 - can request that variable be bound to register

C summary: data abstraction

- variable introduction
 - declaration at start of program
 - declaration at start of block
 - formal parameters
- scope
 - lexical
- extent
 - block

Assembler summary: data abstraction

- names R0–R15, PC, LR, SP etc abstract from CPU registers
- labels abstract from memory addresses
- names and labels used as variables
- i.e. use name as operand to access/change register/memory
- no data structures
 - must craft explicitly from byte sequences

Assembler summary: data abstraction

- registers:
 - scope/extent - whole program
- labels:
 - scope whole file + where imported
 - extent – whole program

Control abstraction

- structured operations as commands
- how are calculations performed?
 - e.g. Java: expression
- how is memory accessed?
 - e.g. Java: use variable name in expression context
- how is memory changed?
 - e.g. Java: assignment to variable

Control abstraction

- how are commands structured?
- e.g. Java:
 - sequence
 - block, nested method calls
 - choice
 - if, switch
 - repetition
 - while, for, iterators, recursion

Control abstraction

- e.g. Java
 - control flow
 - method call, return & break
 - procedural
 - void method
 - i.e. from control sequences
 - functional
 - method with return type
 - i.e. from expressions
 - call by reference parameter passing

Program abstraction

- encapsulation
 - abstract over data & control
- e.g. Java
 - classes/objects

C summary: control abstraction

- expressions
 - abstract from arithmetic/logic sequences
- commands
 - abstract from:
 - memory/register manipulation sequences
 - flag test, branch & address

C summary: control abstraction

- commands
 - assignment
 - sequence
 - block & function body
 - choice
 - if & switch
 - repetition
 - for & while
 - flow of control
 - function call, return, break & goto

C summary: control abstraction

- functions
 - functional abstraction
 - with return type
 - procedural abstraction
 - no return type
 - call by value & by reference parameter passing

Assembler summary: control abstraction

- operators abstract from machine code
- must craft structured constructs from operator sequences
- no universal standards or conventions
 - but compilers/operating will define standards
 - e.g. for parameter passing

Pragmatics

- what is mapping to byte machine?
- how implemented?
- how easy to read/write/debug?
- performance?
- use?
- etc...

C summary: pragmatics

- in-between assembler & high-level
- many to one mapping from command to machine code
- must be compiled to machine code (or interpreted)
- easier to read/write/debug than assembler
- can manipulate low level byte machine
- weak typing → errors from pointers & casts
- must manage memory explicitly
 - e.g. allocate free memory for dynamic structures

C summary: pragmatics

- not as fast as assembler
- CPU independent
 - ISO standard
 - mostly portable
 - but size of type depends on implementation
- used for system level programming, real-time/embedded control
- rapidly replacing assembler for lower-level programming

Assembler summary: pragmatics

- direct machine code abstraction
- 1 to 1 mapping from assembly language to machine code
- easier to program than machine code
- must be assembled to machine code
- very fast
 - same as machine code

Assembler summary: pragmatics

- hard to read/write/debug
- CPU specific
 - not portable
 - 1950s attempts at universal machine code (UNCOL) failed
- used for mission/system critical software
 - e.g. device drivers, BIOS, small embedded systems

Language choice

- formally, programming language choice is arbitrary
- *Church-Turing thesis*
 - all formal models of computability are equivalent
- thesis, not a theorem
 - can't be proved as unknown number of models
- simplest way is to demonstrate *Turing completeness*
 - show model can be used to implement an arbitrary *Turing Machine*
 - Alan Turing's 1936 model for computability

Language choice

- *all* models of computability so far have been shown to satisfy Church-Turing thesis
- including:
 - lambda calculus
 - recursive function theory
 - generalised von Neumann computers
 - programming languages with arbitrarily large values/structures
- so, in principle, can do anything in any programming language
- but...

Technical constraints on language choice

- language may not support key constructs
 - e.g. lower level system/platform manipulation/control
 - e.g. high level control abstractions suitable for parallelism
- language may not be supported on mandated platform
- implementation may not be able to deliver required performance
- implementation may not have particular:
 - functionality
 - libraries
- could fix these but adds time/effort/cost to project

Social constraints on language choice

- may prefer one language over another
 - why do you prefer the language?
 - not necessarily a good basis for choice...
- language may be mandated by:
 - employer
 - customer
 - either may have:
 - substantial investment in specific language technology
 - requirement for specific system/platform/external interoperability

Socio-technical constraints on language choice

- need to be future proof
- language should be:
 - stable
 - widely used
 - widely supported by different platform/language tool manufacturers