SENG2200

Lecture 1 – Introduction

* Programming Domains
  + Scientific applications
    - very large figures, e.g. Fortran
  + Business applications
    - reports, cash figures and characters, e.g. COBOL
  + Web software
    - collection of languages: markup (HTML), scripting (PHP), general purpose (Java)
  + Artificial intelligence
    - manipulate symbols rather than numbers, e.g. Lisp
  + Systems programming
    - efficiency through continuous use, e.g. C
* Language Evaluation
  + Readability
    - how easily someone (including the same developer at a later time) can read and understand the code
  + Writability
    - how easily the code can be used to create programs
  + Reliability
    - conformance to specifications in all situations
  + Cost
    - how much money development costs – most important to businesses/people
    - cheapest in the short-term is not always cheapest in the long-term
* Language Characteristics and Evaluation Criteria
  + Simplicity
    - readability: manageable set of features and constructs
    - not many ways to do the same thing (little multiplicity)
    - minimal operator overloading
    - writability: small numbers of primitives
  + Orthogonality
    - readability: a relatively small set of primitive constructs can be combined in a relatively small number of ways
    - purely orthogonal = all type combinations are possible
    - writability: small number of rules for combining primitives
  + Abstraction support
    - writability: use complex structures or operations in ways that allow details to be ignored
  + Data types
    - readability: allow abstraction or modularisation, and facilities for defining data structures
  + Syntax design
    - readability: meaningful key words, self-descriptive constructs, flexible composition and ways of forming compound statements
  + Expressivity
    - writability: convenience in specifying operations
    - reliability:
  + Type checking
    - reliability: testing for type errors
  + Exception handling
    - reliability: intercept and correct run-time errors
  + Restricted aliasing
    - reliability: whether there are two or more distinct referencing methods for the same memory location
  + Cost evaluation
    - training programmers to use the language
    - writing/compiling/executing programs
    - availability of free compilers
    - poor reliability leads to high costs
    - maintaining programs: fixing errors, extending a program and integrating it into another system
  + Portability
    - ease of moving programs from one implementation to another
  + Generality
    - applicability to a variety of applications
  + Well-definedness
    - completeness and precision of languages official definition
* Influences on Language Design
  + Computer Architecture
    - languages are developed around the prevalent, von Neumann architecture
  + Programming Methodologies
    - New development methodologies (e.g. OOP) led to new programming paradigms and languages
* The von Neumann Architecture
  + Data and programs stored memory, separate from CPU
  + Instructions and data are piped from memory to CPU
  + Basis for imperative languages
    - Efficient iteration
    - Assignment statements model piping
    - Variables model memory cells
* Influences on Programming Methodology
  + 1950s and early 1960s: simple applications, focus on machine efficiency
  + Late 1960s: focus on people efficiency – readability and better control structures
    - top-down design and step-wise refinement
    - structured programming
  + Late 1970s: From procedure-oriented to data-oriented
    - data abstraction
  + Middle 1980s: Object-oriented programming
    - data abstraction + inheritance + polymorphism
* Language Categories
  + Imperative: centred around variables, assignment statements and iteration – e.g. C, Pascal
  + Functional: computations made by applying functions to given parameters – e.g. LISP, Scheme
  + Logic: rule-based – e.g. Prolog
  + Object-oriented: encapsulation and information hiding (data abstraction), inheritance (re-use) and polymorphism (extensibility), e.g. Java, C++, C#, Eiffel, Smalltalk
  + Markup: Specifies the layout of information in Web documents, e.g. X/HTML
* Language Design Trade-Offs
  + Reliability vs. Cost
    - e.g. Java requires proper indexing for all references to array elements but leads to higher execution costs
  + Readability vs. Writability
    - e.g. APL provides many powerful operators allowing for complex computations at the cost of low readability
  + Writability vs. Reliability
    - e.g. C++ pointers are powerful and flexible but not reliably used
* Implementations Methods
  + Compilation – programs are translated into machine language
  + Pure interpretation – programs are interpreted by another program known as an interpreter
  + Hybrid systems – a compromise between compilation and interpretation
* Compilation
  + translate high-level (source code) into low-level (machine language)
  + slow translation, fast execution
  + phases of compilation:
    - lexical analysis: converts characters in the source into lexical units
    - syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of the program
    - semantics analysis: generate intermediate code
    - code generation: machine code is generated
* Execution of Machine Code
  + Fetch-execute cycle (on von Neumann architecture):
    - initialise program counter
    - repeat:
      * fetch instruction pointed to by counter
      * increment counter
      * decode instruction
      * execute instruction
    - end repeat;
* Von Neumann Bottleneck
  + Computer speed = connection speed between a computer’s memory and its processor, limited by von Neumann bottleneck (among other things)
  + von Neumann bottleneck = where program instructions execute faster than the connection speed
* Pure Interpretation
  + no translation, slower execution (10-100 times slower than compilation)
  + easier implementation (immediate display of run-time errors)
  + usually requires more space
  + rare on modern languages
* Hybrid Implementation
  + high-level language is translated to intermediate language for easy interpretation
  + examples:
    - Perl programs are partially compiled to detect errors before interpretation
    - initial implementations of Java were hybrid – the intermediate, bytecode, provides portability to any device with a Java Virtual Machine (JVM), regardless of OS, etc.
* Just-In-Time Implementation Systems
  + translate programs to an intermediate language, then compiles into machine code
    - machine code is kept for subsequent calls
  + JIT systems are used for current Java calls and .NET languages
  + also used in some research languages in attempting to implement “live activation” of updated features
* Pre-processors
  + macros are commonly used to specify that code from another file is to be included
  + process a program immediately before the program is compiled to expand embedded pre-processor macros
  + e.g. C pre-processor, expands #include, #define, etc.
* Programming Environments
  + a collection of tools used in software development
  + Borland JBuilder
    - used for Java development
  + Microsoft VS.NET
    - used for C#, C++, J#, Jscript, Visual BASIC.NET

Lecture 2

Lecture 3

Lecture 4

Lecture 5

Lecture 6 – Iterator and Container Implementation with Generic Specifications

* Containers and Collections
  + Linear:
    - Stacks, queues, deques, lists, arrays, heap, priority queue
    - Have an explicit successor and predecessor item
      * e.g. stack/deque/queue – time of insertion and removal are the important organising features
  + Graph
    - Undirected, directed
  + Hierarchical
    - Heap, trees
  + Unordered
    - Sets, bags, maps
* Iterators and Collections
  + A method of visiting every item in a collection
  + Needs a way of determining which items have been visited
  + Clients should be able to traverse collections and visit any item stored in it
  + Operations: Create iterator, test for new items, look at next item, and maybe remove from container

Lecture 7

Lecture 8

Lecture 9

Lecture 10

Lecture 11