General Concepts

- Abstraction
- Computational Paradigms
- Implementation
- Application Domains
- Influence on Success
- Influences on Design
Abstractions in Programming Languages

- Abstractions hide details that are relevant only to the internal structure of each part.
- Abstractions provide complexity control
Assembly language

- abstract instruction codes to symbols
- abstract memory locations to symbols

High level languages

- Data and Control
- Basic, Structured, Unit, Parallel
• Basic Data Abstractions
  – abstract memory locations to simple variables
    * closely supported by hardware
    * char, integer, real, decimal, boolean

• Structured Data Abstractions
  – Structured variables
    * arrays, records, structs, classes, lists
    * may be built-in or user-defined
– Simple Control Abstractions
  * assignment, jump

– Structured Control Abstractions
  * Guarded Commands
    · nestable sets of instructions ’guarded’
      by tests: if, switch, while,...
  * Procedural Abstractions
    · abstracts several actions into a
      single action
    · provides control over program
      interactions
    · two parts: declaration and in-
      vocation
• Unit Data and Control Abstractions
  – stand-alone part of a program
  – controlled interface
  – data encapsulation
  – information hiding
  – C++ class, Ada package, Modula-2 module, components (Component Object Model, CORBA), containers, libraries

• Parallel Control Abstractions
  – Threads (Java) within Java
  – Processes (Java) outside Java
  – Tasks (Ada)
Computational Paradigms

- imperative
- functional
- logic
- object-oriented
- event-driven
- concurrent
Turing Completeness

A language that has enough mechanisms to describe any computation that a Turing machine can perform is called Turing Complete.
Illustrative Example – GCD

- The greatest common divisor of a and 0 is a.

- The greatest common divisor of a and b is the same as the greatest common divisor of b and the remainder of a divided by b.

- GCD will be computed only for non-negative integers.
GCD – Iterative Version

gcd(aa, bb)
begin
    a = aa; b = bb;
    if (b!=0) then
        repeat
            rem = a mod b;
            a = b; b = rem;
        until (b == 0);
    return(a);
end;
GCD – Recursive Version

gcd(a, b)
begin
if (b = 0)
then
  return a
else
  return gcd(b, a mod b)
end;
Imperative Paradigm

- imperative or procedural languages
- model: von Neumann machine
- Fortran, C, Pascal, Algols
- characteristics
  - control: instruction sequencing, assignment, loops, and selection
  - data: variables that represent memory locations

A programming language is Turing Complete if it has integer variables and arithmetic and sequentially executes statements, which include assignment, selection and loop statements.
Functional Paradigm

• functional or applicative languages

• model: recursive function theory
  – 1940’s: Alonzo Church, $\lambda$-calculus
  – less dependent on machine model
  – easier to reason about behavior

• Lisp, ML, Miranda, FP, Scheme

• characteristics
  – control: function call (basic mechanism) and recursion
  – data: function parameters and function values
A programming language is Turing Complete if it has integer values, arithmetic functions on those values, and a mechanism for defining new functions using existing functions, selection, and recursion.
Logic Paradigm

- logic or declarative languages
- model: symbolic logic
  - far removed from machine model
  - very high level language
- Prolog, SQL, RPG, CLP
- characteristics
  - control: Horn clause resolution
  - data: variables that represent partial computations
Object-Oriented Paradigm

• object-oriented languages
• model: interaction among objects, each of which has its own data and operations
• Simula67, SmallTalk, Eiffel, C++
• characteristics
  – control is provided by operations on objects
  – data is provided by variables in objects
Event-Driven Paradigm

- continuous loop that responds to unpredictable events
- events generated by keystrokes, mouse clicks, or robot sensors
- Visual Basic, Java
Concurrent Paradigm

- collection of cooperating processes
  - SIMD, MIMD
  - share information (messages or shared memory)
  - run asynchronously (synchronization mechanisms needed)

- SR, Linda, HPF, Ada
Implementation

• compilers, interpreters, hybrid systems

• a computation is written in ”source” language

• a computation is carried out in machine language

• implementation is the interface between source code and its execution

• implementation depends on the operating system for resource management, i/o, file management, editors and other functions
• Compilers
  – compiler phases: lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, optimization, code generation
  – compilation followed by linking and loading operations
  – program executes independently from compiler
• Interpreters
  – interpreter provides a software simulation of the hardware
  – program executes under control of the interpreter
  – slow because high-level statements are decoded
  – needs more space
  – better debugging possible
• Hybrids
  - generate intermediate code, as in the compiler
  - interpret intermediate code, as in the interpreter
  - Perl, Java (byte code is the intermediate, interpreted by JVM)
• Preprocessors
  – not actually an implementation method
  – processes commands embedded in source code before compilation
  – expands macros, copies files into the program
Application Domains

- **Scientific**
  - data: floating point, arrays and matrices
  - control: loops and selection
  - efficiency
  - Fortran, Algol 60 and descendants

- **Business**
  - data: characters and decimal numbers
  - control: decimal arithmetic
  - large reports
  - COBOL
• AI
  – data: list
  – control: symbolic computation, creation of code
  – Lisp, Prolog, now C

• Systems
  – control: low level features
  – efficiency
  – PL/S, BLISS, Extended Algol, C

• Web
  – presentation + computation
  – XHTML, Java, JavaScript, PHP
Influences on Success

• Technical Influence: Adherence to Design Goals
  – Fortran - efficiency of execution
  – Cobol - English-like readability
  – Algol60 - block structure for expression of algorithms
  – Pascal - simplicity and promotion of top-down design
• Non-Technical Influences
  – Availability, price and quality of translators
  – Politics, timing, geography, markets

Examples
  – C - promoted by availability of Unix
  – PL/I - supported by IBM
  – Ada - required by DoD
  – Algol
    * popular in Europe
    * IBM supported Fortran in US
    * stack-based machine not well understood in US
Influences on Design

- Computer Architecture
- Programming Methodology
- Implementation Method
- Execution Model
- Programming Environment
- Language Paradigms
- Intended Use
Computer Architecture

• procedural languages based on von Neumann machine

• functional languages
  – generally not so close
  – Lisp machine: TI explorer

• stack languages
  – generally not so close
  – stack machine: Burroughs 5500
Programming Methodology

• should be supported by language
  – Algol 60 – structured programming
  – Simula 67, Ada – data abstraction
  – Pascal top – down design
  – Smalltalk, C++, Java, Python, Ruby – object model
  – Ada – concurrency
  – dataflow
  – Linda, HPF – parallel programming

• new languages evolve as new methodologies develop
Implementation Method

- interpreted or compiled?
- some features are better supported by one method or the other
- features that need to be determined during execution fit interpreters
- features that can be determined before execution fit compilers
Intended Use

• Special Purpose
  – abstractions can be built in
  – often used for general purposes
  – examples
    * rapid prototyping (VB)
    * beginners(logo,basic)
    * database (sql)
    * graphics(logo, VPython)
    * real-time (Ada)
• General Purpose
  – facilities for user-defined abstractions
  – often designed for a particular use
  – examples
    * C - system programming
    * Lisp - symbolic computation
    * Fortran - numerical computing
    * Ada - concurrency and real-time programming