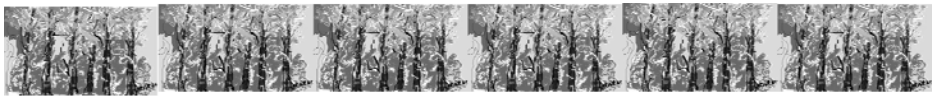


Chapter 1 – Introduction
Introduction to Programming Languages
First Edition, 2013

Author: Arvind Bansal
Kent State University
Kent, OH 44242

Topics Covered

- **Topics covered**
  - Multitude of problem domains
  - Learning outcomes
  - Program components
  - Interoperability of programming languages
  - Criterion for good Languages
  - Software development cycle
  - Programming paradigms and history
  - Programming language classifications
  - Summary

Problem Domains

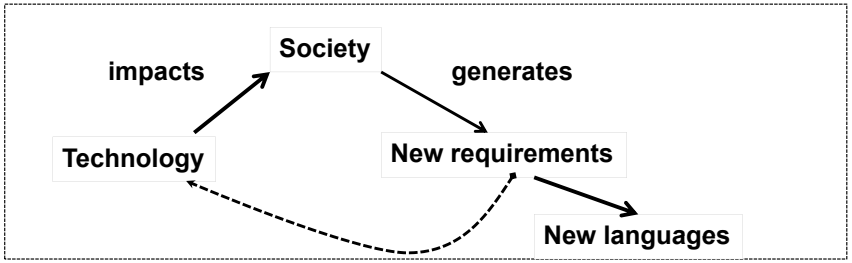
- **Scientific computing**
  - Large matrices, excessive memory, needs efficient computation
- **Text processing**
  - String processing capability, user friendliness
- **Database programming**
  - Search and file management capability; fast indexing capability
- **Business applications**
  - Report generation, decimal number specification
- **System programming**
  - Memory manipulation, fast execution, low level instructions
- **Real time processing**
  - High priority to real-time tasks, quick decision making
- **Intelligent systems**
  - Symbolic computation, heuristic and probabilistic reasoning
- **Web based applications**
  - Portability across machines, visualization, quick data conversion

Motivation

- **Automating real-world processes**
  - Requires high level specification of solution.
  - Programs are organized symbols to communicate and instruct computers in meaningful way.
  - Computers are low level machine incapable of understanding human intentions.
- **Programming language is a way to organizing meaningful symbols**
  - Symbols can be textual, voice, visual, gestures, or multimodal.
  - Instructions could be static, dynamically created, and/or mobile.
  - Instructions should have unique meaning and express programmer's intention unambiguously for correct solution.
  - Language supports ease of expression and program evolution.
  - **Meaningful:** intended action is the same as actual action

Evolution of Languages

- **Technology, society and language requirements are intricately related**
  - As the technology evolves, it impacts the society that fuels new requirements.
  - New requirements mean new ways of solving problems, and new ways to express the solutions.
  - New languages and new technology

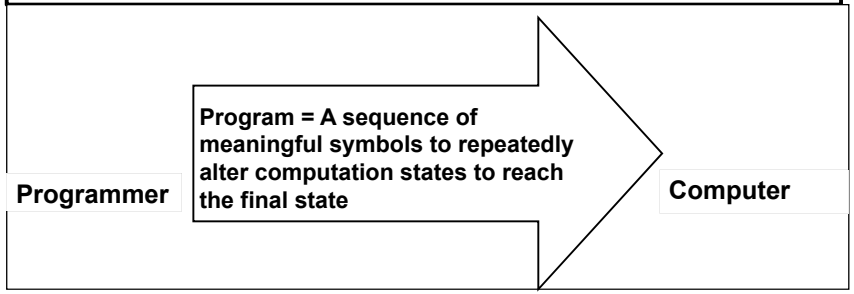


Learning Outcomes

- **Shorter learning curve or new languages**
  - Deeper understanding of paradigm abstractions
  - Help in understanding multi-paradigm languages of future
- **Awareness of low level behavior of program constructs**
  - Avoid pitfalls of programming errors
  - Improve the development of efficient programs
- **Background for the development of compilers**
  - Understanding low level behavior is essential for code generation.
- **Improvement in programming style**
  - Learn constructs and abstractions from different programming styles
  - Avoid programming pitfalls
- **Better selection of languages for specific problems**

Program Components

- **Program**
  - Specifications of a solutions to handle dataflow between modules
  - A sequence of meaningful symbols instructing computer
- **Components of program**
  - Logic - high level specification of solution
  - Abstraction – modeling entity by desired common attributes
  - Control – mapping solution on von-Neuman machine where an instruction progressively changes the state of computation



Example of Program Components

- **Problem: Sorting an unsorted bag of numbers**
- **Abstraction: identifying entity by desirable attributes**
  - Model bag as an indexible sequence
- **Logic: stepwise formal specification of solution**
  - Exchange the adjacent number if a[i] > a[I + 1];
  - Perform exchanges for all the adjacent pairs in the sequence starting from the first element;
  - Leave the last element in the subsequence;
  - Repeat the process until one element is left.
- **Control: progressive change of computational state**
  - Map indexible sequence on memory locations
  - Destructively update memory location during exchanges

Program Organization

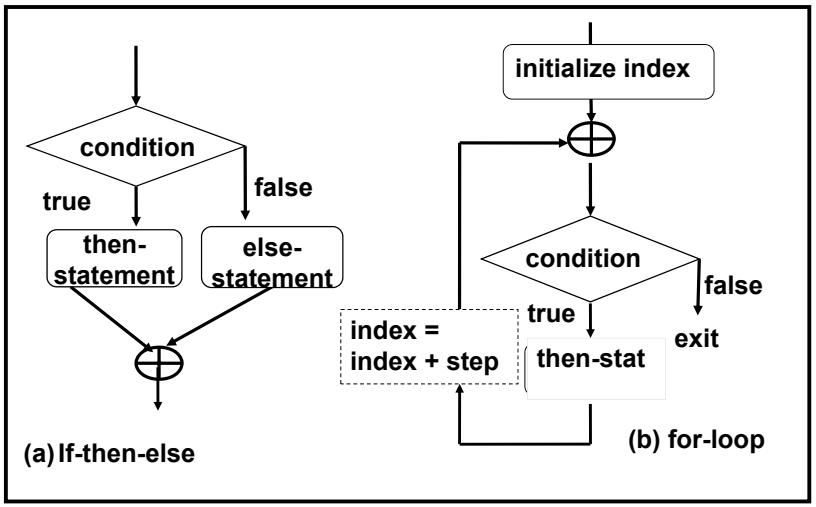
- **Components**
  - Program name
  - Importing previously developed software library
  - Parameters - exchange data with other program units
  - Local environment - data types and variables
  - Body - where computation is done
- **Needs code structuring for better comprehension**
- **Needs unique meaning for structures for one-to-one mapping to low level machine code**
- **Needs modularity for software evolution and maintenance**

Abstractions in Programs

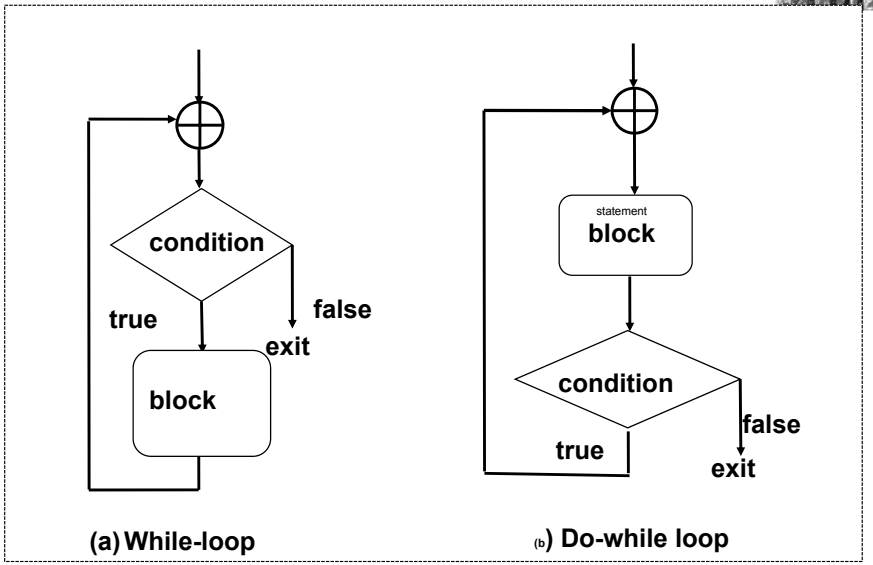
- **Two types of abstractions**
  - Data abstractions and control abstractions
  - Data abstractions are abstraction related to modeling real-world entities using a subset of abstract attributes
  - Control abstractions are abstractions related to grouping program statements using some common properties such as if-then-else, for-loop, while-loop etc.

```
const class_size = 20;
    struct student {
          string student-id;
          string student-name;
          char letter-grade;
          }
student class[class_size];
```


Control Flow Diagrams



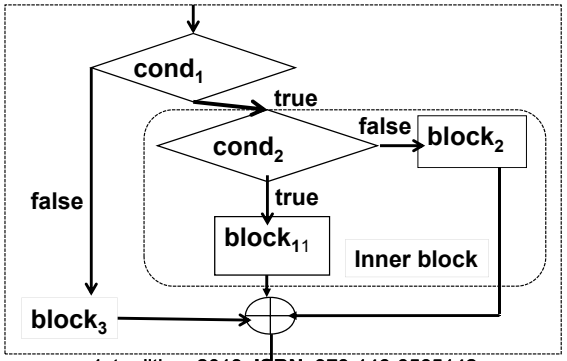
Indefinite Iteration



Making Control Flow Diagrams

- **Technique**
  - Start from the outermost control abstraction;
  - Treat inner blocks as a single block;
  - Make the control flow diagram for the outermost abstraction;
  - Progressively expand inner blocks.

```
if (<cond2>) {
  if (<cond1>
      if (<cond2>)
          <block1>
    else <block2>
  else <block3>
```

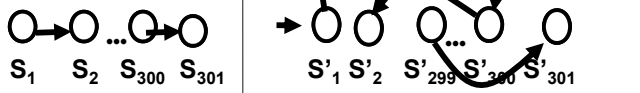


Software Maintenance

- **Developed software needs maintenance because**
  - Need evolves
  - Programmers move on
  - Programmers forget their code and approach
  - Technology changes, and software needs to be ported
- **Program maintenance needs**
  - Enough comments
  - Better program comprehension using structured programming
  - More abstraction
  - Simplicity of abstraction

Program Comprehension

- **Structured programs**
  - Programs have mostly forward flow with little jumps.
  - Jumps are embedded inside control abstractions
  - Jumps are only used to exit out of nested blocks
- **Criteria for better Program comprehension**
  - Background knowledge and training in programming
  - Level of abstractions provided in a program
  - Simplicity of code and abstractions
  - Structured programs, easily categorized, modular in function



a. structured program

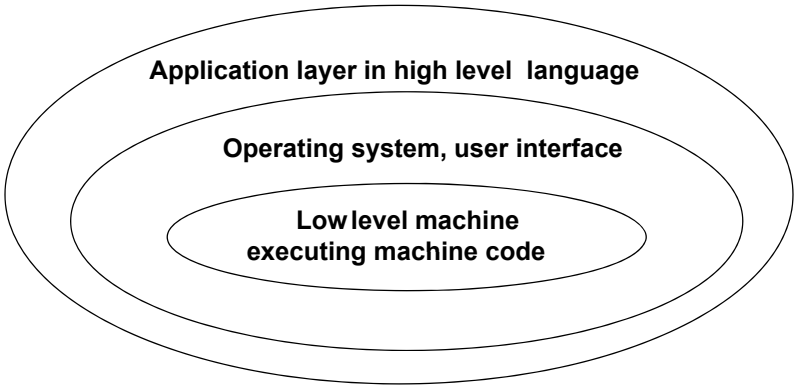
b. program with jumps

Criteria for Sound Execution

- **The language constructs of a programming language should be well defined.**
- **There should be a unique meaning for every language construct as computers do not handle ambiguities.**
- **Each high level instruction should be translated to a sequence of low level instructions that perform the same action consistently on a computer every time.**
- **A computer should execute consistently the same sequence of low level instructions producing the same final result.**

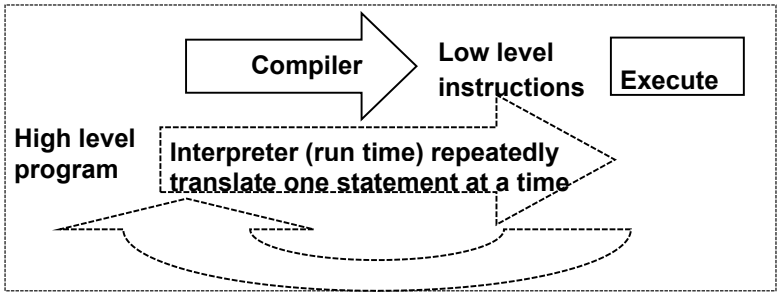
Layers of Software in Computer

- Inner most layer: low level machine code
- Middle layer: operating system and user interfaces
- Uppermost layer: application layer



Translating High Level Instructions

- **Compiler**
  - Static translation and then execute; No runtime overhead
- **Interpreter**
  - Translate and execute one statement at a time – runtime overhead; an order of magnitude slower
- **Just-in-time compilation (to be discussed)**



Compiler vs. Interpreters

Compiler

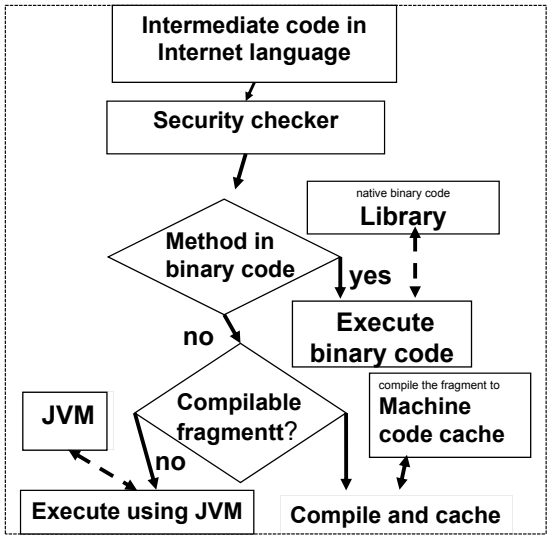
- Static translation
- No runtime overhead
- Faster by an order of magnitude
- Many times not sufficient technology to compile all languages
- Problem is compiling mobile code
- Just-in-time compilation for web-based languages

Interpreter

- Interleaved runtime translation and execution
- Runtime overhead
- Slower by an order of magnitude
- Easy to develop
- Support execution of dynamically typed languages and mobile code
- Use abstract machine for portability
  - better machine independence

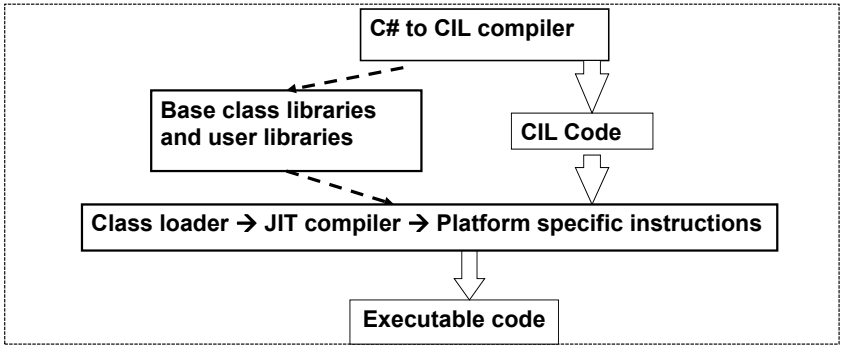
Just-in-time Compilation

- **Approach**
  - Compile at runtime once and cache the compiled code
  - Execute if already in binary code library or in cache
- **Runtime compilation overhead**
- **Performance between interpreter and compiler**



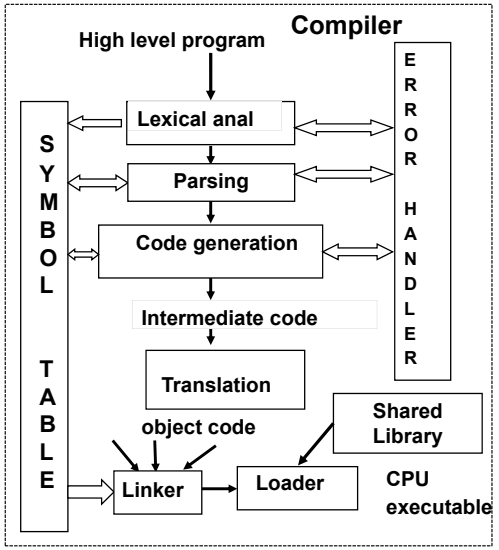
Just-in-time Compilation of C#

- **Uses a common interface language CIL**
- **Transforms C# to CIL**
- **Uses JIT and built-in class loader to translate to executable code**



Compiler Stages

- **Lexical analyzer**
  - Converts into tokenized format
- **Parser**
  - Checks validity of sentence structure using grammar
  - Outputs in tree structure
- **Code generator**
  - Generates instructions into intermediate code
  - Optimizes the code
- **Translator**
  - Converts intermediate code to object code
- **Linker** links multiple code



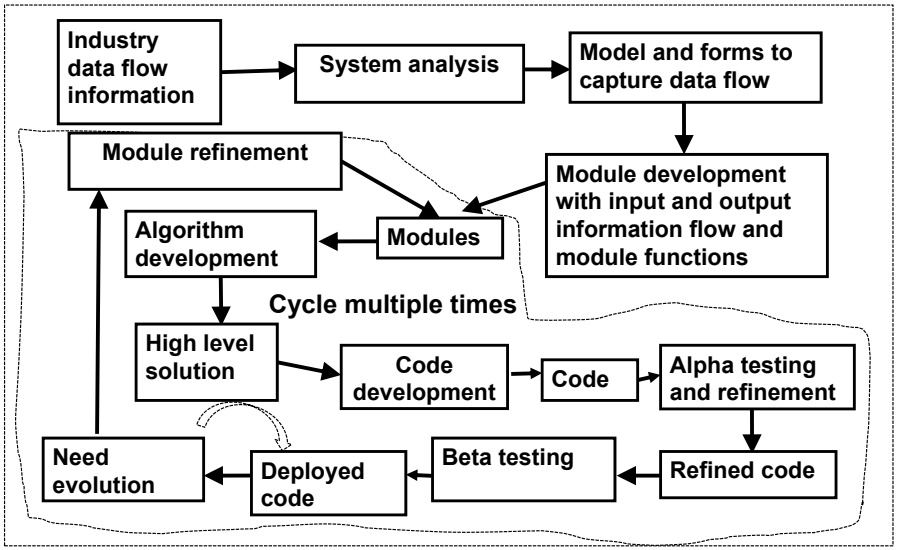
Interoperability

- **Ability of code fragment in one language to interact with code fragment of another language**
- **Need**
  - Different problems have more than one problem domains needing more than one paradigm and language.
  - Improve efficiency by interfacing with libraries in low languages.
  - Maximize code reuse by using libraries in other languages.
- **Mechanism**
  - Exchange information using shared metadata – a separately declared data being shared between two languages
  - Convert back-and-forth from data-structure to meta-data, or
  - Common language specification and standard common data type
  - Use common types across multiple languages
  - Common mechanism to store and retrieve data types in across languages

Software Development I

- **A large software can be millions of lines of code**
  - May involve more than one problem domains and more than one programming paradigm and language.
  - Requires a team of programmers, system analysts, field testers.
- **Stages in life cycle of software**
  - Requirement analysis, system analysis and design, development, implementation, initial verification, field testing, deployment, and incremental evolution
  - Requires feedback at every stage to refine the previous stage
- **Requirement analysis:** knowledge gathering from client
- **System analysis:** Analyze scope and model information flow
- **Design phase:** Information flow is split in modules and linkages
- **Development phase:** Data structures, algorithms and interfaces
- **Software testing :** Testing the usefulness for the problem
- **Software modification and maintenance :**
  - Modular (factorized), portable, easy to comprehend; Evolution against new requirements, change in technology, change in the developmental team

Software Cycle Schematic



Criteria for a Good Language

- **Abstraction**
- **modularity**
- **Orthogonality – language constructs independent**
- **Exception handling – get out of error condition gracefully**
- **User-friendliness – better interfaces**
- **Readability – self explanatory variables and comments**
- **Ease of comprehension and maintenance**
- **Overall simplicity**
- **Portability: moving application to different architectures**
- **Efficient execution**
- **Requirements are contradictory**

Requirements are Contradictory

- **Abstraction**
  - Formal specification
  - Structured
  - Better comprehension
  - Ease of maintenance: modular routines; locality of features
  - Slow execution
- **Declaration free**
  - Ease of programming
  - Run time program failure
- **Efficiency**
  - Jump statements
  - Machine dependent
  - Difficult to understand
- **Compile time declaration**
  - Difficult to program
  - Fast execution
  - Compile time error detection
  - Memory optimization

Programming Paradigms I

- **Imperative programming**
  - Mutable variables destructive update, explicit user control, reusability, side-effects, past lost
- **Declarative programming**
  - Immutable assign-once variables, transparent control, memory explosion, less side-effect, first class objects & meta-programming
  - logic programming and functional programming used for AI
- **Object-oriented programming**
  - Modularity, inheritance, templates, information hiding
- Concurrent and distributed programming
  - Synchronization and independent subtasks
- **Visual programming – drag and drop screen based**
- **Web based programming**
  - Code and data mobility, visualization, interaction
  - Multiple resources, user-friendliness, security
  - Multiple modality of visualization: text, image, video etc.

Programming Paradigm II

- **Multimedia based programming**
  - Multimodal visualization – text, sound, image, video, gesture and their integration
  - Synchronization of media streams
- **Event based programming**
  - Actions based upon events rather than procedures
  - Used popularly for graph and web based interaction
- **Paradigm Integration**
  - Modern languages integrate more than one paradigm
  - C# and Java : event based + multimedia + imperative + web based + object oriented + concurrent programming
  - C++ : imperative + object oriented
  - Paradigm integration can be built in for by interfacing with a library

History

- **1960s - Large scientific computation**
  - Fortran – subroutines, goto statements
  - ALGOL 60 - supported recursion and recursive data types, block structured processing, text processing, abstract types
- **1970s - Text processing, portability, and modularity**
  - Low level languages for system programming based on ALGOL
  - High level languages such as Pascal, Modula, PL-1 etc.
  - Functional programming language Lisp used for AI
- **1980s - Modularity, software and concurrency**
  - Logic programming language Prolog
  - Concurrent languages such as Parallel Fortran, Concurrent C
  - Object oriented programming paradigm: C++, Eiffel, Smalltalk
- **1990s: Web, Multimedia, event and paradigm integration**
  - Languages: Java, JavaScript, PhP, SMIL, C#, Alice,

Evolution of Languages

- **Multiple old languages evolved**
  - Integrating newer constructs and paradigms to take care of the previous limitations
- **Examples**
  - Fortran 66 → Fortran 77 → Fortran 90 → Fortran 2008
  - Ada 85 → Ada 93 → Ada 2005 → Ada 2012
  - Cobol has a revised version Cobol 2002
  - C → C++ → Java → C#
- **Modifications**
  - Fortran got block structuring, object oriented programming
  - Cobol 2002 supports object oriented programming
  - Paradigm shift from C to C++ to Java and C#

Classification of Languages

- **Implementation based classification**
  - Static allocation: upfront memory allocation, no runtime growth
  - Stack allocation: runtime growth, recursive procedures
  - Heap allocation: heap is a common shared space. supports recursive data types: trees, lists, dynamic objects, overhead of memory of allocation & recycling
  - Integrated allocation: use all three appropriately
- **Type based classification**
  - Monomorphic vs polymorphic type: polymorphism allows same function or operator on multiple data types
  - Statically vs dynamically typed: dynamically typed variable is bound to data type at runtime based upon assignment
- **Paradigm based classification: already discussed**
- **Modern day languages use all the useful properties, and boundaries have become fuzzy**

Summary

- Program = logic + control + abstraction
- Two types of abstractions: control abstraction and data abstraction
- Programming language is an organized way of communicating with computers using unambiguous meaningful symbols
- Languages evolve based upon technology and society needs
- Criteria for good programming languages are often contradictory
- Multiple programming paradigms are there: imperative programming, declarative programming, object oriented programming, concurrent programming, visual programming, web and multimedia programming, event based programming etc.
- Modern day languages support multiprogramming paradigm
- There are three types of memory allocations: static, stack based and heap based. Modern languages integrate all three allocations
- Languages can use monomorphic types, polymorphic types, or a combination of both. Languages are statically typed or dynamically typed