Programming Languages

Dr. Prakash Raghavendra

- Mtech (IIT Madras, CSE, 1992), PhD (IISc, CSA, 1998)
- Hewlett-Packard, Bangalore: 1998-2007
  - Kernel, Manageability, Compilers and Java
- Adobe Systems, Bangalore: 2007-2009
  - Flex Profiler
- NITK, Surathkal, Assoc Prof: 2009-2012
- AMD, Bangalore, 2012-Date
  - OpenCL, HSA Compilers for GPUs
  - Java optimization for Server

Course Plan

- Instructor: Prakash Raghavendra
- Lecturers:
  - Friday: 2 hours
  - Saturday: 2 sessions of 2 hours each
  - Two weeks a month
- Evaluation:
  - One quiz (10 marks)
  - One assignment (10 marks)
  - One mid-sem (30 marks)
  - One end-sem (50 marks)

Course Materials

- Textbook:
- *Robert W.Sebesta, "Concepts of Programming Languages", 9th Edition, 2009*
- *Ravi Sethi, "Programming Languages - concepts and constructs", Addison Wesley, 2nd Edition, 1996.*
  - Attend lectures! Lectures will cover some material that is not in the textbook – and you will be tested on it!

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation
5-10	Paradigms, Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Functional PLs, Logical PLs, Lambda Calculus, Event driven programming, Concurrency
29-36	Virtual Machines, Managed Languages, JIT, Case study

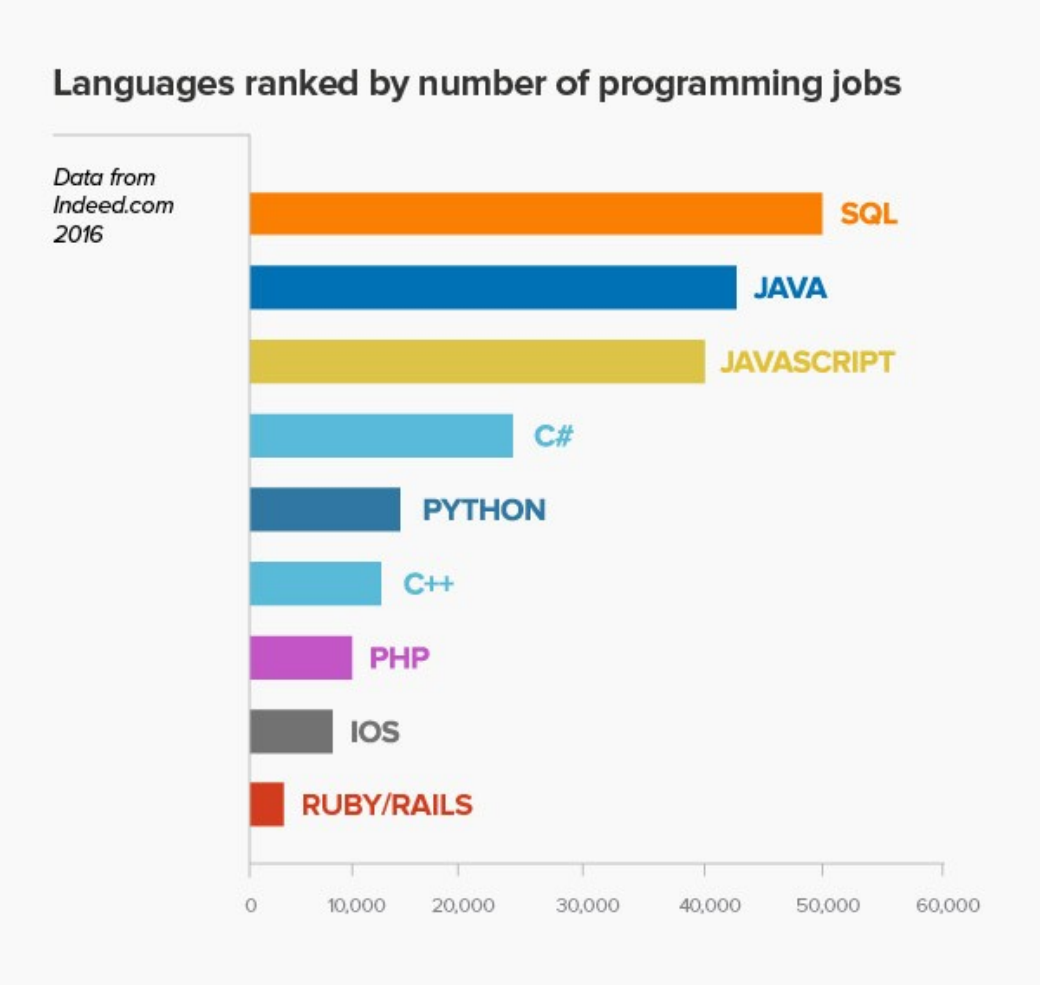
Some Course Goals

- Language as a framework for problem-solving
  - Understand the languages you use, by comparison
  - Appreciate history, diversity of ideas in programming
  - Appreciate the implementation of the languages
- Critical thought
  - Identify properties of language, not syntax or sales pitch
- Language and implementation tradeoffs
  - Every convenience has its cost
    - Recognize the cost of presenting an abstract view of machine
    - Understand tradeoffs in programming language design

What's Worth Studying?

- Dominant languages and paradigms
  - C, C++, Java… JavaScript?
  - Imperative and object-oriented languages
- Important implementation ideas
- Performance challenges
  - Concurrency
- Design tradeoffs
- Concepts that research community is exploring for new programming languages and tools

Popular Languages



Latest Trends

- Commercial trends
  - Increasing use of type-safe languages: Java, C#, …
  - Scripting and other languages for Web applications
- Teaching trends: Java replacing C
- Research and development trends
  - Modularity
  - Program analysis
    - Automated error detection, programming environments, compilation
  - Isolation and security
    - Sandboxing, language-based security, …

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Programming Domains

- Scientific applications
  - Large numbers of floating point computations; use of arrays
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)
- Heterogeneous Programming
  - OpenCL, OpenACC, CUDA

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications)
- **Cost**: the ultimate total cost

Evaluation Criteria: Readability/Writability

- Overall simplicity
  - A manageable set of features and constructs
  - Minimal feature multiplicity
    - For example, in C++ or Java you can decrement a variable in four different ways: `x = x – 1; x -= 1; x--; --x`
  - Minimal operator overloading
  - Some languages (e.g. assembly languages), can be "too simple" – too low level. 2, 3, 4, 5 or more statements needed to have the effect of 1 statement in a high-level language
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
  - C: function cannot return a static array (or assemble instructions)
  - However, if a language is too orthogonal, an inexperienced programmer might assume they can do something that makes no sense, e.g. add two pointers together

Evaluation Criteria: Readability/Writability..2

- Structured programming improves readability/writability
  - The following are equivalent

```
if (x < y) x++;        if (x < y) goto L1;
 else y++;              y++;
                      goto L2;
                    L1: x++;
                    L2:
```

- Data types
  - Adequate predefined data types

Eg: A language with Boolean types is easier to read than one without

  - indicatorFlag = 0

is more difficult to read than

  - indicatorFlag = false

Evaluation Criteria: Readability/Writability..2

- Syntax considerations
  - Syntax - the way linguistic elements (e.g. words) are put together to form phrases or clauses/sentences
  - Identifier forms
    - If too short, reduces readability
  - Special word use
    - Ada has end if and end loop, while Java uses } for both
    - In Fortran 95, Do and End can also be variable names
  - Form and meaning
    - In C, static changes meaning depending on position

Abstraction

- The ability to define and then use complex structures or operations
  - Allows details to be ignored
  - Allows code to be re-used instead of repeated
  - Example: A binary tree in Fortran 77 required arrays, while in OO languages, nodes with pointers may be used

1. Abstract data types
   - implementation details are separated from the interface, allowing them to be changed without re-writing all code
2. Objects
3. Subprograms

Abstraction Increases Expressivity

- Expressive language - has powerful built-in primitives for high-level abstractions
- For example, in Lisp
  - Pointer manipulation is implicit – avoid mistakes
  - Mapcar – apply a function to every element of a list (and return the corresponding results in a list)
    - No need to write the iteration yourself – you would need to write a different function for each different type of data
- Infinite precision integers and rational numbers
  - No need to develop functions yourself
  - Completely avoid round-off errors at will
    - E.g. 2/3 + 1/3 = 1, not .999999

Evauation Criteria: Reliability

- A reliable program **performs to its specifications under all conditions**
- Factors that affect reliability
  1. Type checking
  2. Exception handling
  3. Aliasing
  4. Readability and writability
  5. Environmental factors – real-time or safety-critical application?

Type Checking and Exception Handling Improve Reliability

- Type checking
  - Testing for type errors in a given program
    - For example, if a function is expecting an integer receives a float instead
- Exception handling
  - Used in Ada, C++, Lisp and Java, but not in C and Fortran
    - E.g. the try and catch blocks of C++ can catch runtime errors, fix the problem, and then continue the program without an “abnormal end”

Aliasing Reduces Readability and Reliability

- Aliasing
  - Referencing the same memory cell with more than one name
    - E.g., in C, both **x** and **y** can be used to refer to the same memory cell

```
int x = 5;
int *y = &x;
```

  - Leads to errors
- Reliability increases with better read/writability
  - If a program is difficult to read or write, its easier to make mistakes and more difficult to find them

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Evaluation Criteria: Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language's official definition

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

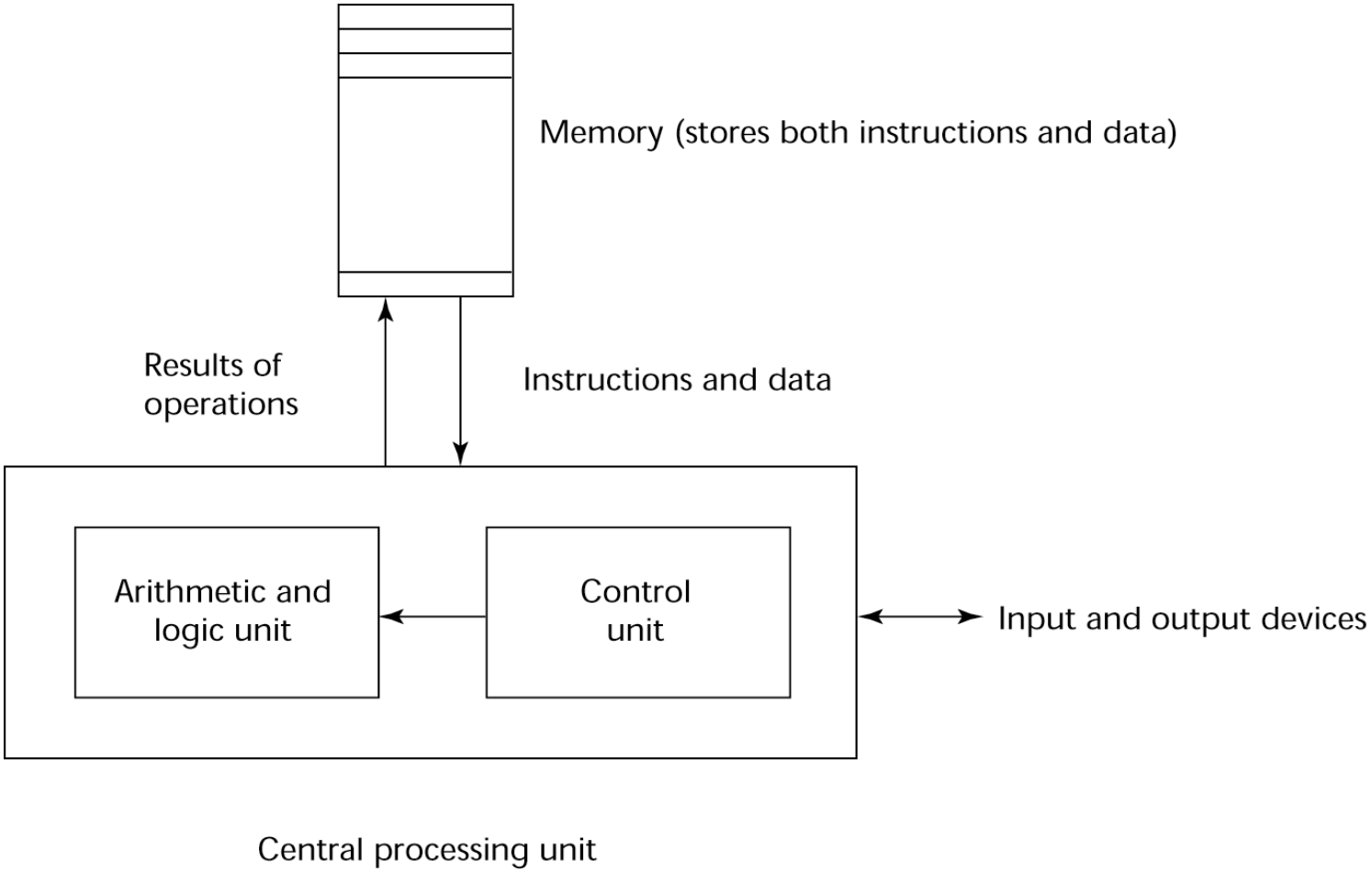
Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Program Design Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

The von Neumann Architecture



The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism
- After 2005: Heterogeneous devices programming

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme, ML, F#

```
(defun factorial (N)
   "Compute the factorial of N."
   (if (= N 1) 1
     (* N (factorial (- N 1)))))
```

Language Categories

- Rule-based (rules are specified in no particular order)
- Example: Prolog

Facts	**English meanings**
food(burger).	// burger is a food
food(sandwich).	// sandwich is a food
food(pizza).	// pizza is a food
lunch(sandwich).	// sandwich is a lunch
dinner(pizza).	// pizza is a dinner
Rules	
meal(X) :- food(X).	// Every food is a meal OR Anything is a meal if it is a food
Queries / Goals	
?- food(pizza).	// Is pizza a food?
?- meal(X), lunch(X).	// Which food is meal and lunch?
?- dinner(sandwich).	// Is sandwich a dinner?

Language Categories

- Markup/programming hybrid
  - Markup languages extended to support some programming
  - Examples: XHTML, MXML (Action Script)

```
<mx:Button id="btn" label="MyButton" height="100" />

var btn:Button = new Button();
btn.label = "MyButton";
btn.height = 100;
```

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability

  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Implementation Methods

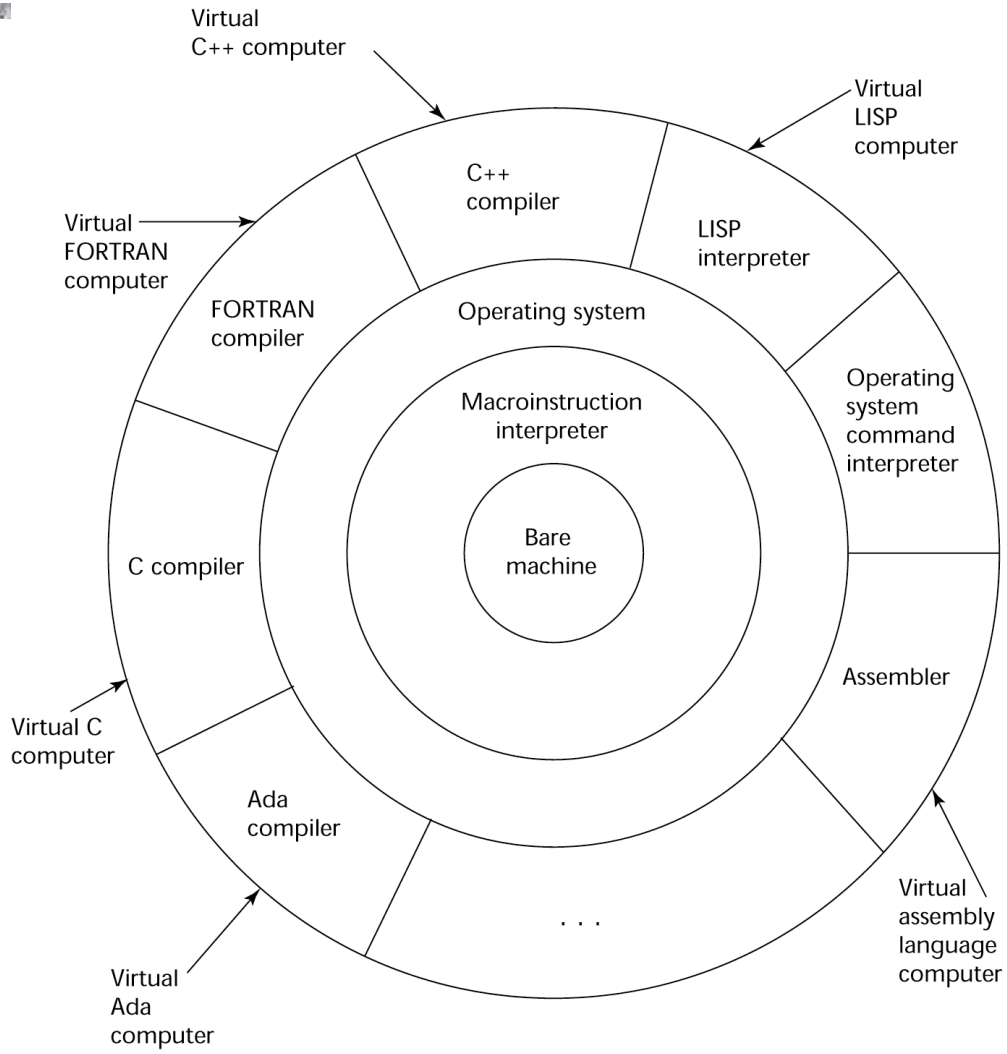
- Compilation
  - Programs are translated into machine language; includes JIT systems
  - Use: Large commercial applications

- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue

- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters
  - Use: Small and medium systems when efficiency is not the first concern

Layered View of Computer

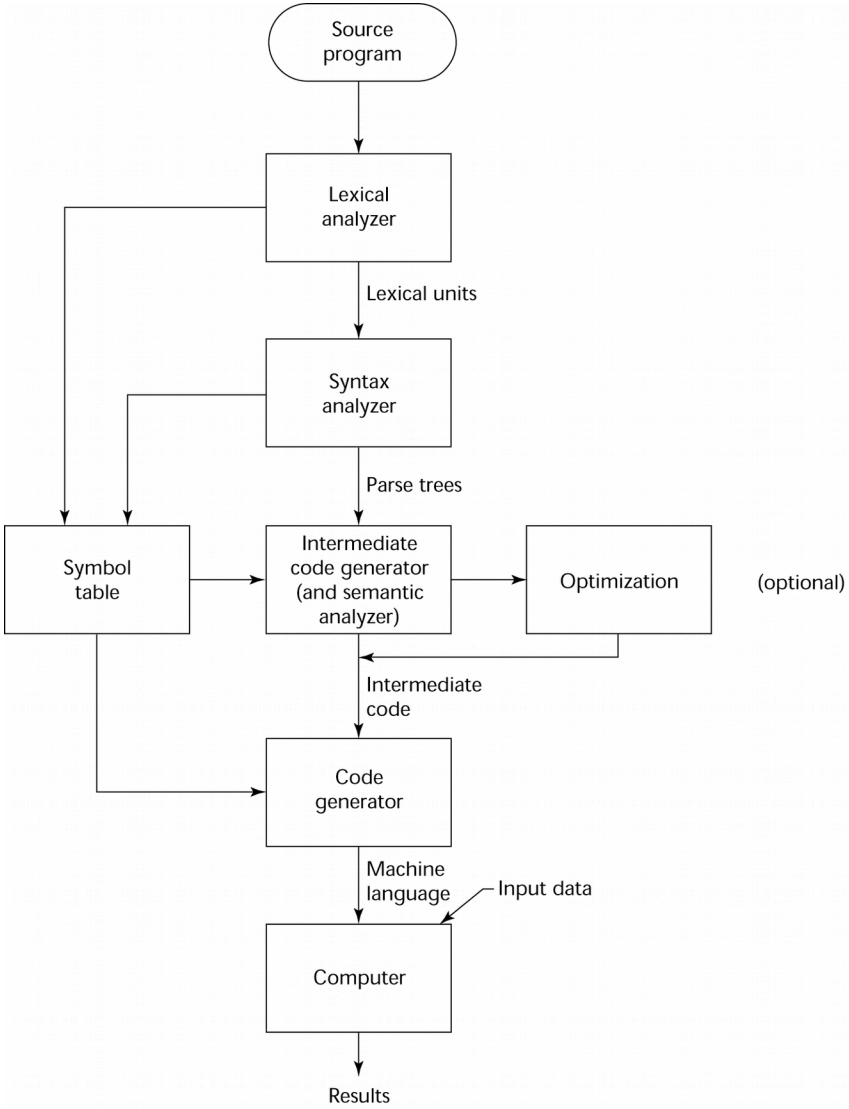
The operating system and language implementation are layered over machine interface of a computer



Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

The Compilation Process



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

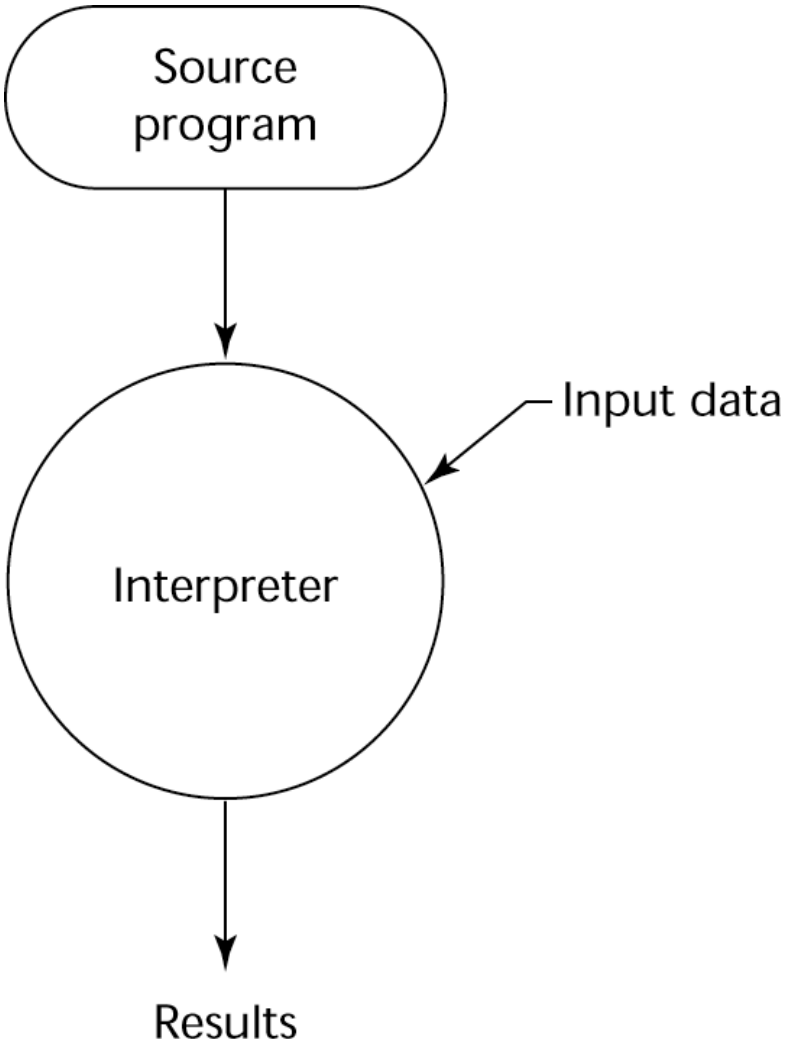
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

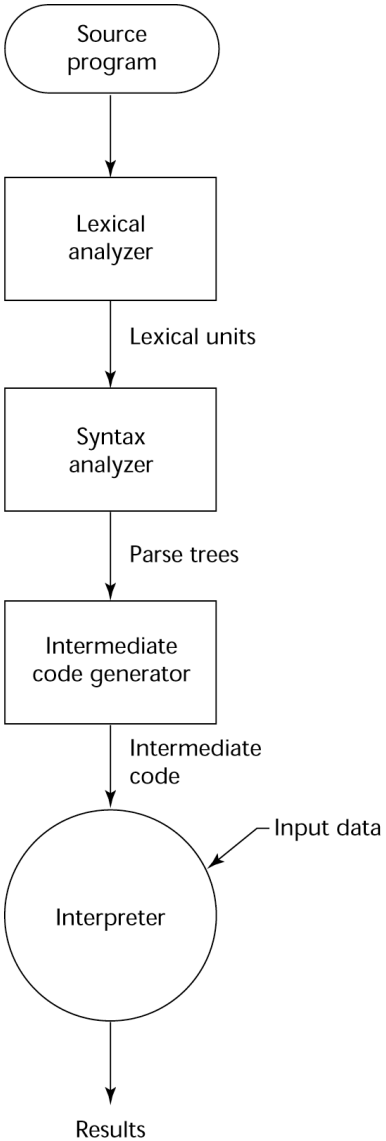
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- In essence, JIT systems are delayed compilers

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands `#include`, `#define`, and similar macros

Objectives

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Programming Environments

- A collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
- Microsoft Visual Studio.NET
  - A large, complex visual environment
- Used to build Web applications and non-Web applications in any .NET language
- NetBeans
  - Related to Visual Studio .NET, except for applications in Java

Summary

- The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation