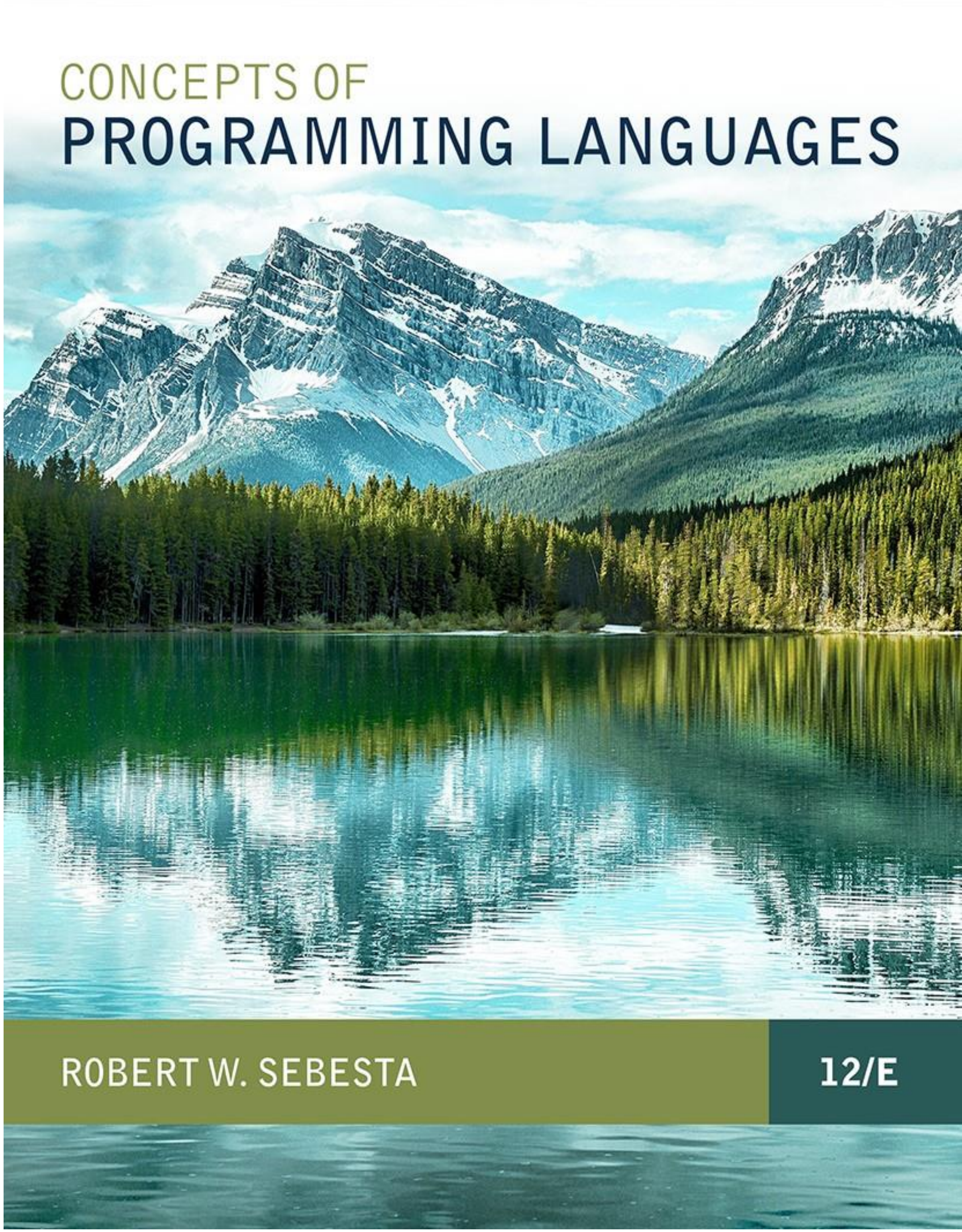
Chapter 1

Preliminaries



ISBN 0-321-49362-1

Chapter 1 Topics

- 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

Reasons for Studying Concepts of Programming Languages

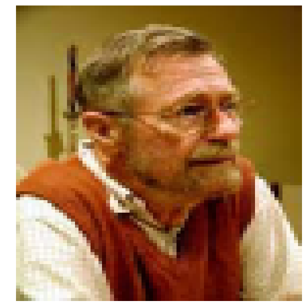
- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Motivation: Why Study Programming Languages?

- **Improves ability to express ideas in primary language**
  - Languages influence the way you think and approach problems
  - As you study new language features it may help you utilize or extend your own language skills
  - Simulate a useful feature in your primary language

Motivation: Why Study Programming Languages?

On language and thought (2)



The tools we use have a profound (and devious!) influence on our thinking habits, and therefore, on our thinking abilities.

-- Edsger Dijkstra, *How do we tell truths that might hurt?*, http://www.cs.umbc.edu/331/papers/ewd498.htm

Edsger Wybe Dijkstra (11 May 1930 -- 6 August 2002), http://www.cs.utexas.edu/users/EWD/

Professor Edsger Wybe Dijkstra, a noted pioneer of the science and industry of computing, died after a long struggle with cancer on 6 August 2002 at his home in Neunen, the Netherlands.

1

5

Motivation: Why Study Programming Languages?

- **Improved background for choosing appropriate languages**
  - Helps you understand the trade-offs in languages rather than immediately assuming your known language is the best one for the job
  - Gives you the background to communicate to others in a logical way the choices necessary to make an informed language decision

Motivation: Why Study Programming Languages?

- **Increased ability to learn new languages**
  - There is significant similarity in the constructs provided by languages so that learning a language is often just a matter of syntax.
    - Selection (if and case)
    - Loops (while, for, do)
    - Jumps (goto, break, continue)
    - Data types (strict or loose, int, char, string, object)
    - Functions

Motivation: Why Study Programming Languages?

- **Helps you understand the significance of implementation**
  - Most things don't happen by chance, there is often a reason behind the way a language was built.
  - Some implementation issues are obviously related to technology.
    - Hardware influences
  - Many aspects of a language are related to softer issues.
    - Who built it and the way the language was promoted
    - Programmer understanding of the value of certain constructs
    - State of the industry
      - Does it need the language? Is it ready for the language?

Motivation: Why Study Programming Languages?

• **Increased ability to design new languages**

– You probably will be designing a language of some sort sometime in your career

Unlikely a full blown programming language, but maybe an XML schema or full markup language DTD, a mini-scripting language for controlling a system, a configuration file language to control software, a simple API/language for data interchange, and so on.

– Just because you can, it doesn't mean you should

Real language value is often very much related to number of people using it.

Motivation: Why Study Programming Languages?

- **Overall advancement of computing**
- Extremely useful for understanding compilers

Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Fortran still alive
- Business applications
  - Produce reports, use decimal numbers and characters
  - Reporting as well as calculations
  - Cobol, reporting languages (e.g. Crystal Reports), scripting environments of business systems like SAP, Siebel, etc.
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - Natural language, string manipulation, and logic needs
    - LISP family (Common Lisp, Scheme, ML), Prolog

Programming Domains

- Systems programming
  - Need efficiency because of continuous use
  - Speed! Safety can be a problem
    - Machine level -> assembly -> C

- Scripting languages
  - Put a list of commands in a file to be executed
  - Generally domain specific
  - Usually interpreted
    - JavaScript, Excel macros, sh, csh, awk, etc.

- Special-purpose languages

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

Language Evaluation Criteria

- Readability
  - Readability describes the ease of which programs can be read and understood.
  - The most important criterium
  - Factors:
    - Overall simplicity
      - Too many features is bad
      - Multiplicity of features is bad
      - count = count + 1; count += 1; count++; ++count;
    - Operator overloading can be trouble
      - • 5 + 6
      - • 5.5 + 6.1
      - • “test” + “it”
      - • “test” + 5
      - • [5, 6, 1] + [1, 3, 4] = [6, 9, 5] or [5,6,1,1,3,4] or 20?

Language Evaluation Criteria

- Readability factors (continued)
  - Control statements
  - Defining data types and structures
  - Syntax considerations
    - Identifier forms
    - Special words
    - Form and meaning

Language Evaluation Criteria

- **Orthogonality** : small number of primitive constructs combined in a relatively small number of ways to build the control and data structures of the language
  - Makes the language easy to learn and read
  - Meaning is context independent
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
  - Lack of orthogonality leads to exceptions to rules

Evaluation Criteria: Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Minimal feature multiplicity
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Data types
  - Adequate predefined data types
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Language Evaluation Criteria

- Writability : the measure of how easily a language can be used to create programs for a given domain.
  - Factors:
    - Simplicity and orthogonality
    - Support for abstraction
    - Expressivity
- Reliability : Reliable programs work under all conditions
  - Factors:
    - Type checking
    - Exception handling
    - Aliasing
    - Readability and writability

Evaluation Criteria: Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Strength and number of operators and predefined functions

Evaluation Criteria: Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support "natural" ways of expressing an algorithm will require the use of "unnatural" approaches, and hence reduced reliability

Reliability

The Explosion of the Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of **development costing $7 billion.** The destroyed rocket and its **cargo were valued at $500 million**. A board of inquiry **investigated the causes of the explosion** and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

The internal SRI software exception was caused during execution of* ***a data conversion from 64-bit floating point to 16-bit signed integer value****. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*

*SRI stands for Système de Référence Inertielle or Inertial Reference System.

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

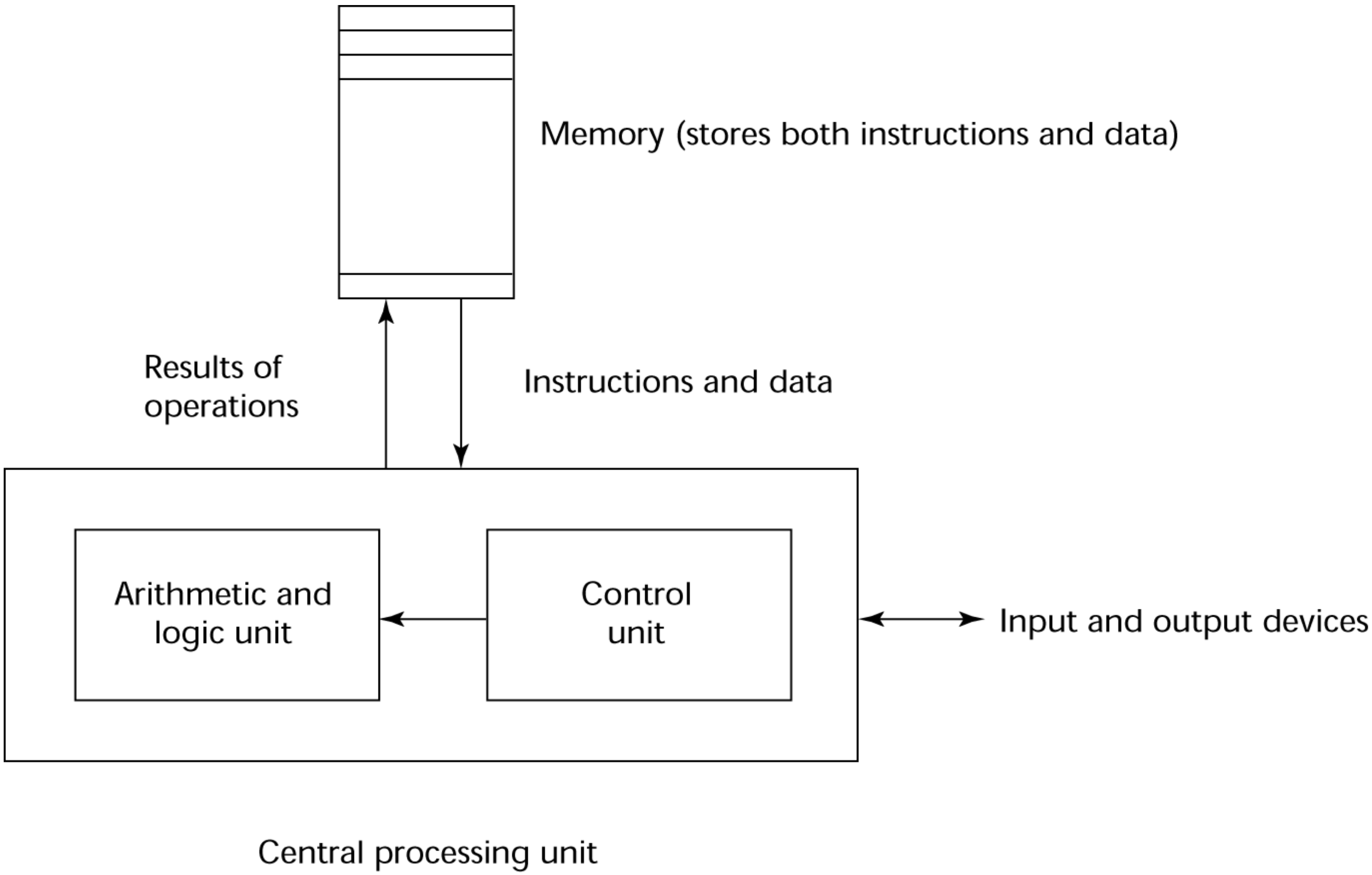
Influences on Language Design

- 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

- The hardware really does influence the software
- The standard computer architecture (von Neumann machine) pretty much dominates language design
- – John von Neuman is generally considered to be the inventor of the "stored program" machines – the class to which most of today's computers belong.
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Focus on moving data and program instructions between registers in CPU to memory locations
- We use imperative languages, at least in part, because we use von Neumann machines
  - 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

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#
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Object-Oriented
  - Encapsulate data objects with processing
  - Inheritance and dynamic type binding
  - Grew out of imperative languages
  - C++, Java
- Markup/programming hybrid
  - Markup languages extended to support some programming
  - Examples: JSTL, XSLT

Language Categories

- Imperative : traditional sequential programming (passive data, active control). Characterized by variables, assignment, and loops.
  - Central features are variables, assignment statements, and iteration
  - C, Pascal
- Functional : passive data, but no sequential control; all action by function evaluation ("call"), particularly recursion. No variables!
  - Main means of making computations is by applying functions to given parameters
  - LISP, Scheme

Imperative Programming Example

```
function gcd(u, v: in integer) return integer is
        y, t, z: integer;
begin
    z := u;
     x := v;
      loop
           exit when y = 0;
           t := y;
           y := z mod y;
           z := t;
       end loop
     return z;
end gcd;
```

- Written in Ada

Imperative Programming Example

```
#include <stdio.h>
int gcd(int u, int v) /* “functional” version */
{  if (v == 0) return u;
   else return gcd (v, u % v); /* “tail” recursion */
}
main() /* I/O driver */
{ int x, y;
  printf("Input two integers:\n");
  scanf("%d%d",&x,&y);
  printf("The gcd of %d and %d is %d\n",
                              x,y,gcd(x,y));
  return 0;
}
```

Functional Programming Example

$$n! = 1 \qquad \text{if } n = 1$$

$$n! = n * (n - 1)! \quad \text{if } n > 1$$

We can implement a function to compute factorials using recursion:

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

Language Categories

- Logic : Assertions are the basic data; logic inference the basic control. Again, no sequential operation
  - Rule-based
  - Rules are specified in no special order
  - Prolog
- Object-oriented : data-centric, data controls its own use, action by request to data objects. Characterized by messages, instance variables, and protection
  - Encapsulate data objects with processing
  - Inheritance and dynamic type binding
  - Grew out of imperative languages
  - C++, Java

Logic Programming Example

```
gcd(U, V, U) :- V = 0
gcd(U, V, X):- not (V = 0)
                Y is U mod V,
                gcd (V, Y, X)
```

- This example was written in Prolog.
- Logic programming is sometimes called *declarative* programming because you declare or make assertions, but no execution sequence is specified.

Object Oriented Example

```cpp
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

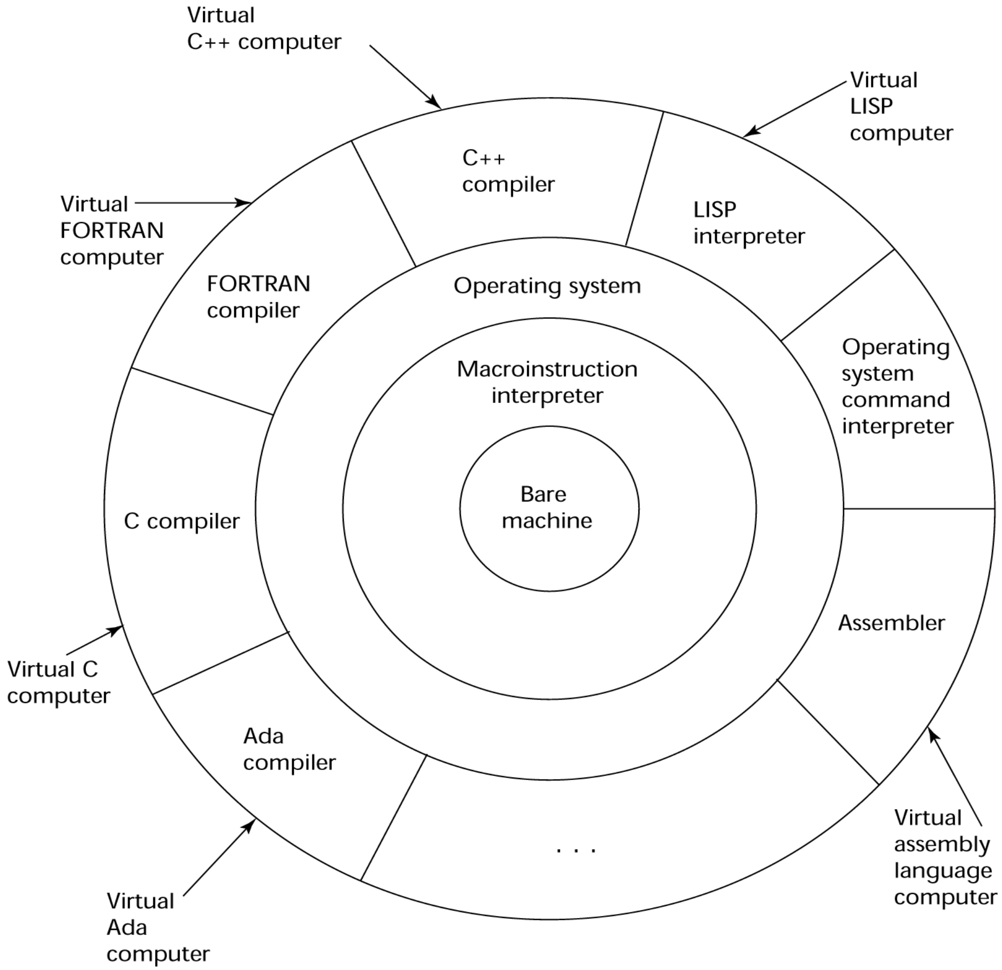
```java
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

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

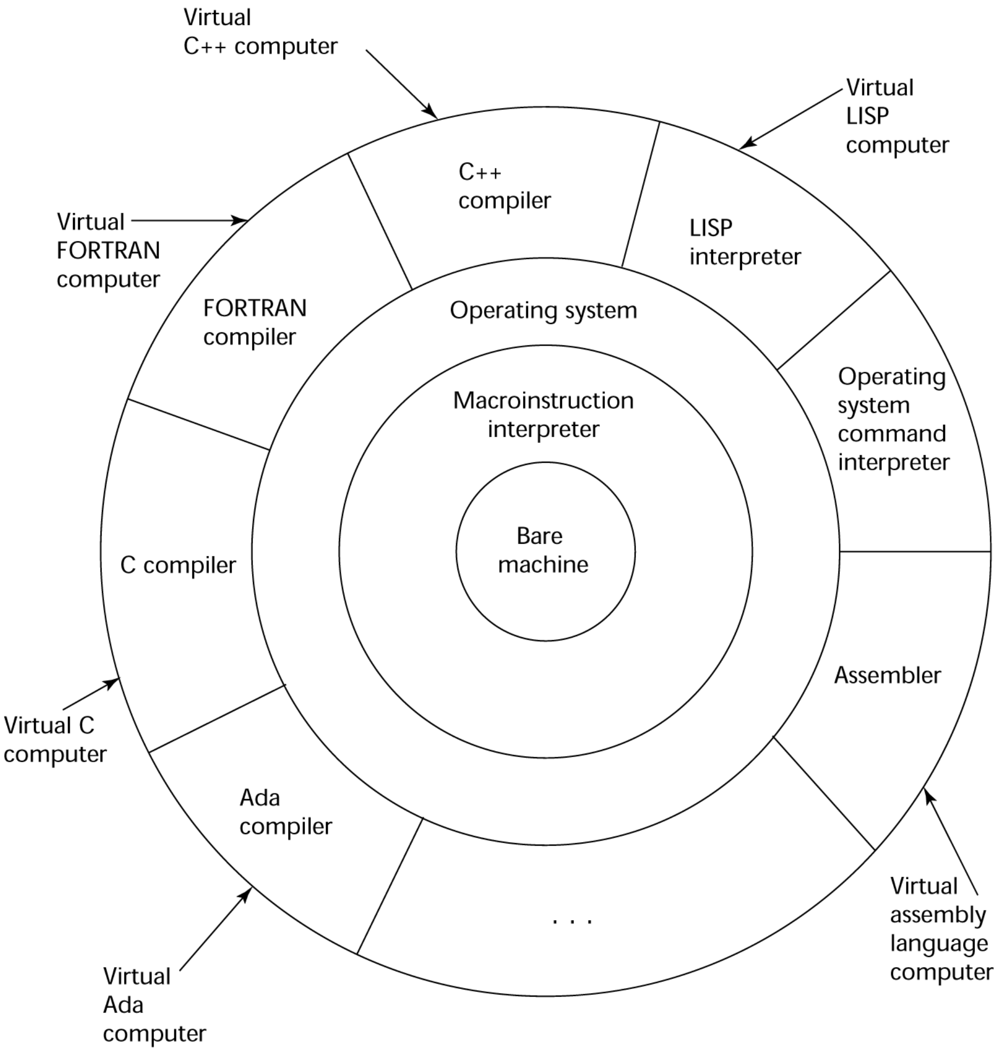
  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

Layered View of Computer



Layered View of Computer

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



The machine and assembly language

- Machine languages consist of a set instructions that computers execute. They consist entirely of numbers and are almost impossible for humans to read and write.
- Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.
- An assembler is a program that translates programs from assembly language to machine language.
- Each type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another.
- In the early days of programming, all programs were written in assembly language.
- Programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a high-level language.

Intel Pentium Assembly

```
TITLE   Add individual digits of a number   ADDIGITS.ASM
COMMENT |
        Objective: To find the sum of individual digits of
                   a given number. Shows character to binary
                   conversion of digits.
            Input: Requests a number from keyboard.
|          Output: Prints the sum of the individual digits.
.MODEL SMALL
.STACK 100H
.DATA
number_prompt  DB  'Please type a number (<11 digits): ',0
out_msg        DB  'The sum of individual digits is: ',0
number         DB  11 DUP (?)

.CODE
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  number_prompt  ; request an input number
        GetStr  number,11     ; read input number as a string
        nwln
        mov     BX,OFFSET number  ; BX := address of number
        sub     DX,DX         ; DX := 0 -- DL keeps the sum
repeat_add:
        mov     AL,[BX]       ; move the digit to AL
        cmp     AL,0          ; if it is the NULL character
        je      done          ;  sum is done
        and     AL,0FH        ; mask off the upper 4 bits
        add     DL,AL         ; add the digit to sum
        inc     BX            ; increment BX to point to next digit
        jmp     repeat_add    ;  and jump back
done:
        PutStr  out_msg
        PutInt  DX            ; write sum
        nwln
        .EXIT
main    ENDP
        END   main
```

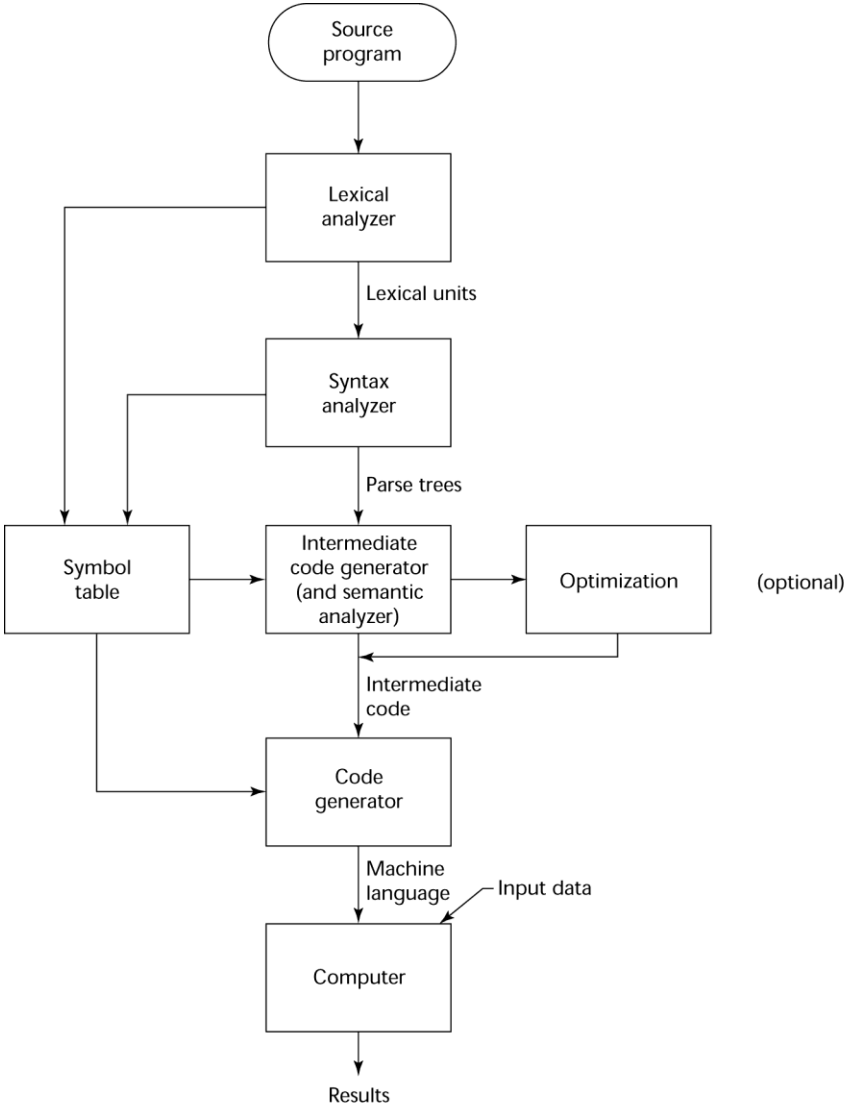
Programming Language Implementation Methods

- 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

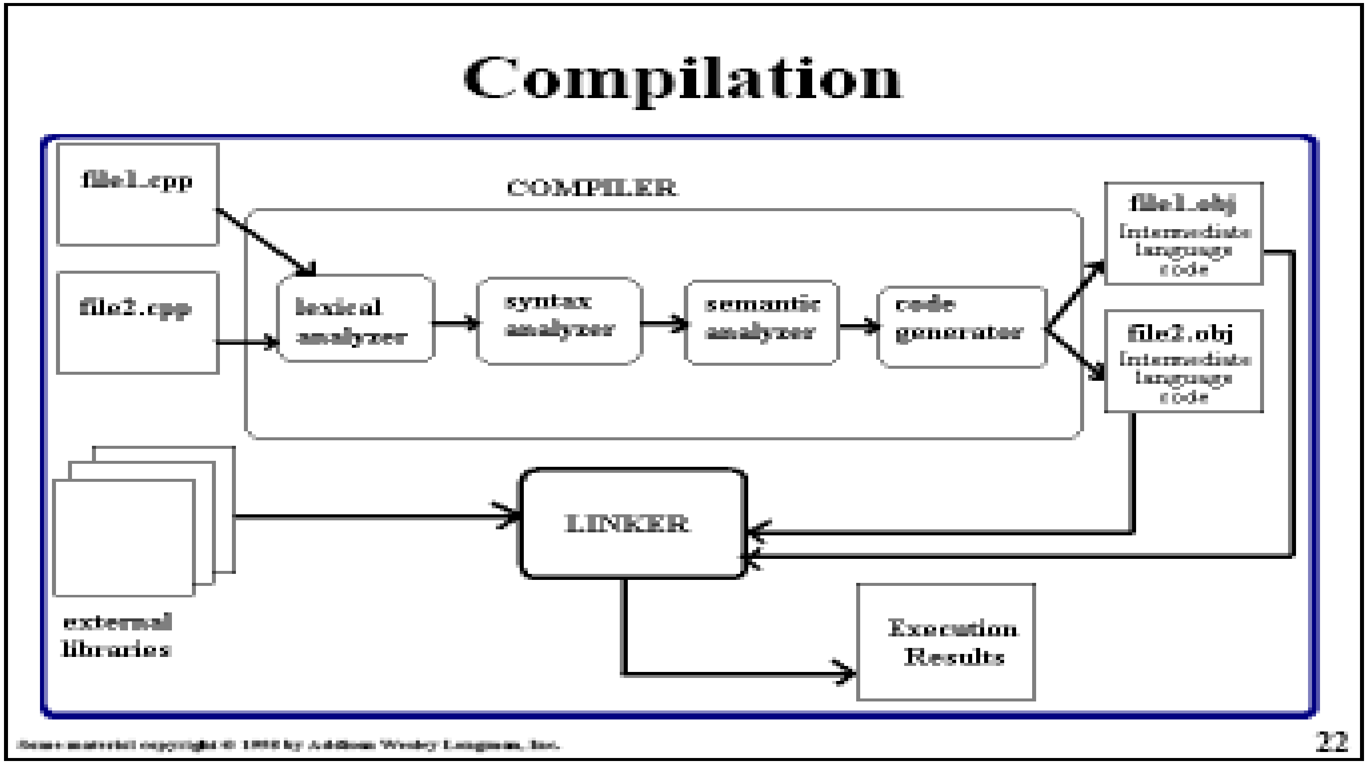
Compilation

- 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

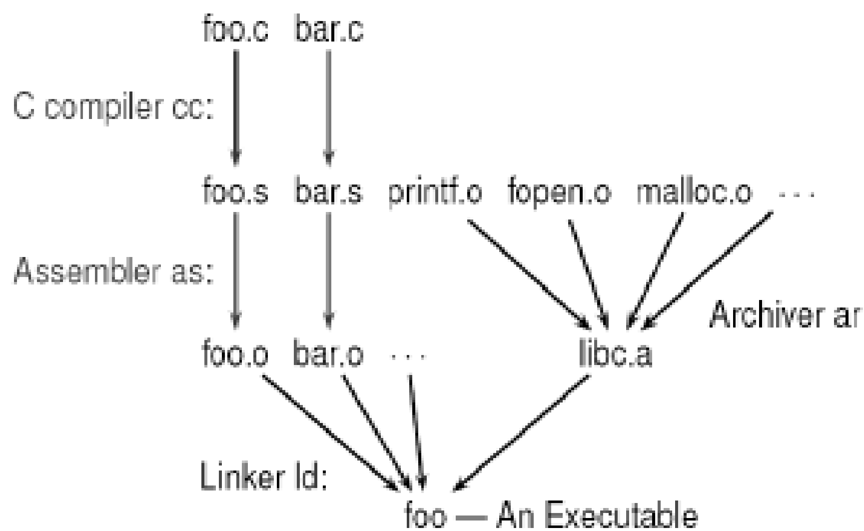


Compiler and Linker



Additional Compilation Terminologies

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



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

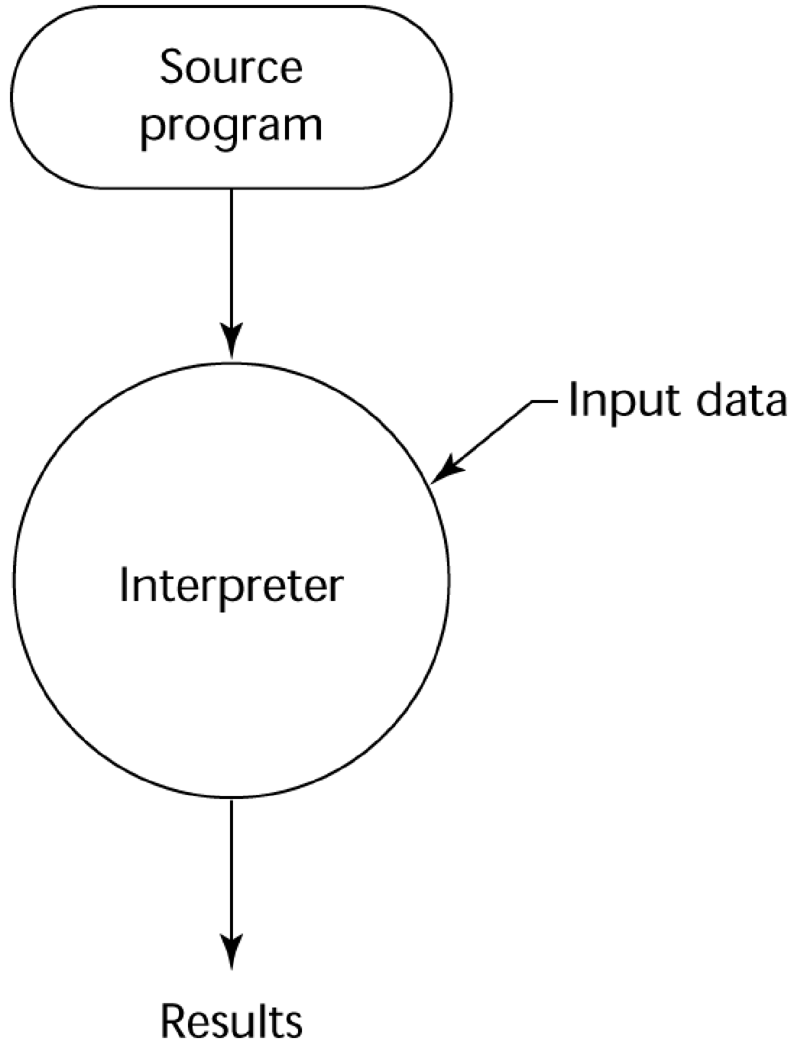
Implementation Methods

- Pure interpretation: program is executed by software
  - No translation
  - Slow execution
  - Becoming rare

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)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

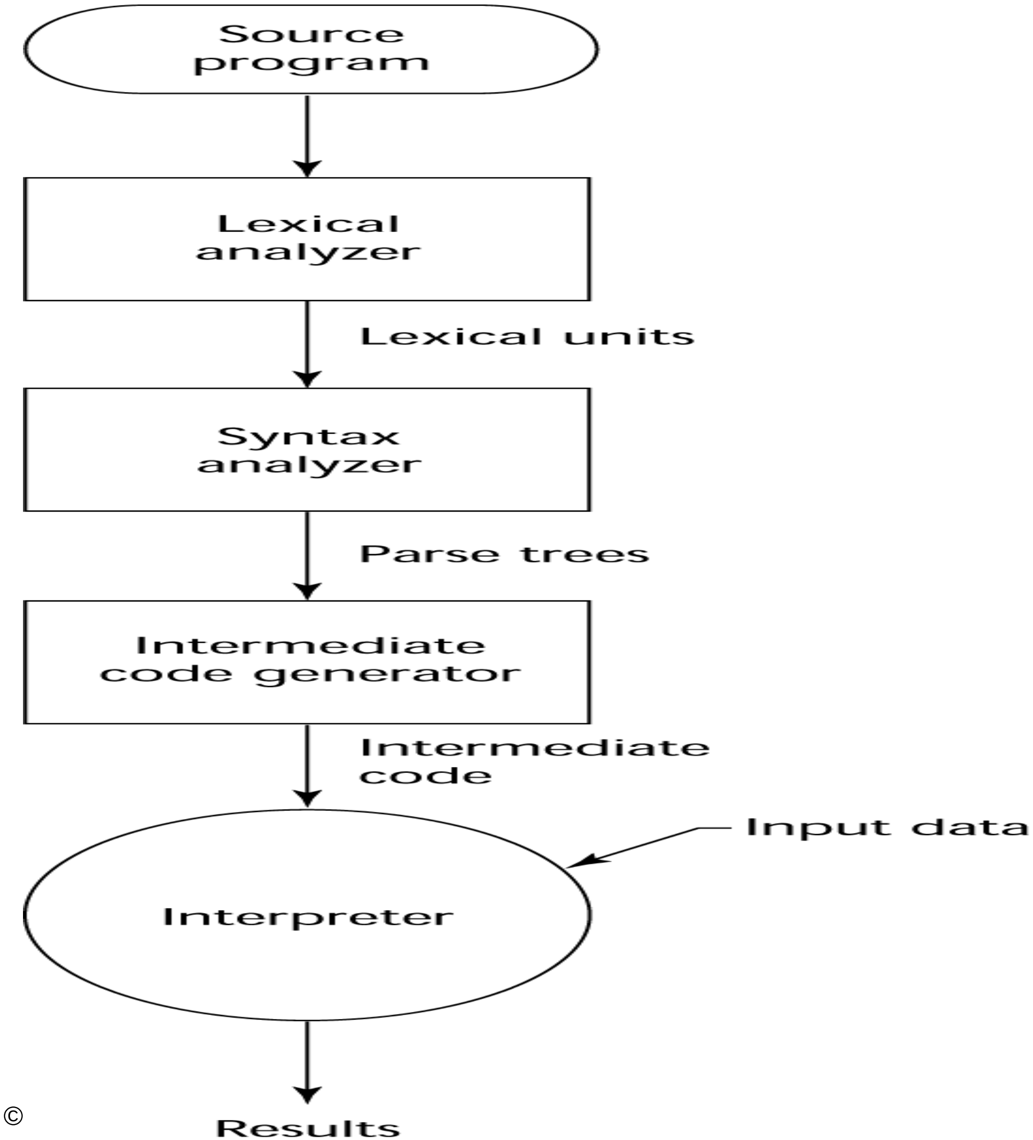
Pure Interpretation Process



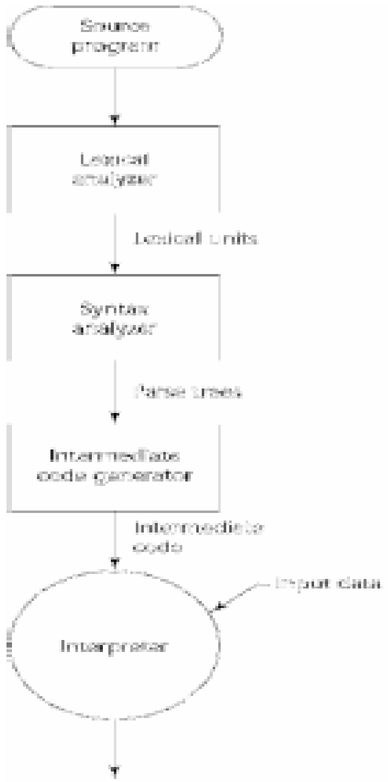
Hybrid Implementation Systems

- 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



Hybrid Implementation Process

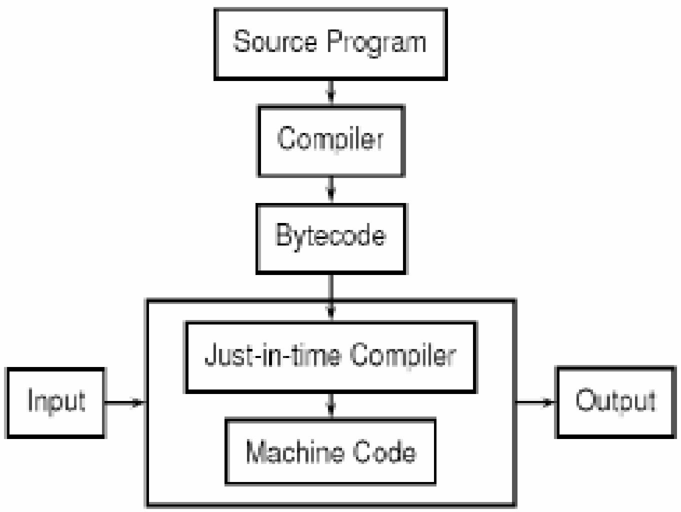


Javac (compiler)	C# (.NET compiler)
↓	↓
Bytecode	MSIL
↓	↓
Java (JVM)	.NET CLR (common language run-time)

Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- "*Just in Time*" (JIT) compilation systems are widely used for Java programs and .NET languages

Just-in-time Compiler



Source:Daniel Ortiz-Arroyo

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

Preprocessors

The example code tests if a macro `__unix__` is defined. If it is, the file `<unistd.h>` is then included. Otherwise, it tests if a macro `_WIN32` is defined instead. If it is, the file `<windows.h>` is then included.

```
#ifdef __unix__ /* __unix__ is usually defined by compilers targeting Unix systems */

# include <unistd.h>

#elif defined _WIN32 /* _WIN32 is usually defined by compilers targeting 32 or 64 bit Windows */

# include <windows.h>

#endif
```

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