

Organization of Programming Languages

CS320/520N

Lecture 01

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

What is an algorithm?

- An **algorithm** is a computational procedure that takes values as *input* and produces values as *output*, in order to solve a well defined *computational problem*:
 - The statement of the problem specifies a desired relationship between the *input* and the *output*.
 - The algorithm specifies how to achieve that *input/output* relationship.
 - A particular value of the *input* corresponds to an instance of the *problem*.

What is a programming language?

- A **programming language** is an artificial language designed for expressing algorithms on a computer:
 - Need to express an *infinite* number of algorithms (Turing complete).
 - Requires an unambiguous **syntax**, specified by a *finite* context free grammar.
 - Should have a well defined compositional **semantics** for each syntactic construct: *operational vs. axiomatic vs. denotational*.
 - Often requires a practical implementation i.e. **pragmatics**:
 - Implementation on a real machine vs. virtual machine
 - *translation vs. compilation vs. interpretation*.

Outline

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

Reasons for studying concepts of PLs

- Increased ability to express ideas/algorithms
 - Natural language:
 - The depth at which people can think is influenced by the expressive power of the language they use (also Sapir-Worf hypothesis).
 - Programming languages:
 - The complexity of the algorithms that people implement is influenced by the set of constructs available in the programming language.
 - Kenneth E. Iverson (inventor of APL programming language):
 - “Notation as a tool of thought” (Turing award lecture).

Reasons for studying concepts of PLs

- Improved background for choosing appropriate languages:
 - Many programmers use the language with which they are most familiar, even though poorly suited for the new project
 - Ideal: use the most appropriate language.
 - Features important for a given project are included already in the language design, as opposed to being simulated => elegance & safety.

Reasons for studying concepts of PLs

- Increased ability to learn new languages:
 - Once fundamental concepts are known, new languages are easier to learn (recognize same principles as incorporated in the new language).
 - Example:
 - Knowing the concepts of Object Oriented Programming (OOP) makes learning Java significantly easier.
 - Knowing the grammar of your native language makes it easier to learn another language.

Reasons for studying concepts of PLs

- Better understanding of significance of implementation:
 - Example: a small subprogram that is called very frequently can be a highly inefficient design choice.
 - More details can be learned by studying compiler design.
- Better use of languages that are already known.
- Overall advancement of computing:
 - Algol 60 vs. Fortran.

Outline

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

Application Domains

- Scientific applications
 - Large numbers of floating point computations; extensive use of arrays
 - Fortran, Matlab
- Business applications
 - Produce reports, use decimal numbers and characters
 - COBOL
- Artificial intelligence
 - Symbols rather than numbers manipulated; use of linked lists
 - LISP
 - Lately, many AI applications (e.g. statistical or connectionist approaches) are written in Java, C++, Python.

Application Domains

- Systems programming:
 - Need efficiency because of continuous use.
 - C (UNIX almost entirely written in C).
- Web Software:
 - Eclectic collection of languages:
 - markup (e.g., XHTML).
 - scripting for dynamic content:
 - *client side*, using scripts embedded in the XHTML document. Examples: Javascript, PHP.
 - *server side*, using the Common Gateway Interface. Examples: JSP, ASP, PHP.
 - general-purpose, executed on the Web server through CGI. Example: Java, C++, ...

Outline

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

Criteria for Language Evaluation

- **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 (e.g. program correctness).
- **Cost:** the ultimate total cost associated with a PL.

Readability

- Overall simplicity
 - A manageable set of basic features and constructs.
 - Minimal feature multiplicity.
 - Example: increment operators in C
 - Minimal operator overloading.
 - Example: C++ vs. Java
- Orthogonality
 - A relatively small set of primitive constructs that can be combined in a relatively small number of ways.
 - Example: addition in assembly on IBM mainframe vs. VAX minicomputers.
 - Every possible combination is legal
 - Example: returning arrays & records in C.

Readability

- Control statements
 - The presence of well-known control structures
 - Example: *goto* in Fortran & Basic vs. *for/while* loops in C.

“Goto” Version:

```
i = 0;
loop1:
  if (i >= 10)
    goto out1;
loop2:
  if (j >= 10)
    goto out2;
  A[i,j] = 1;
  j++;
  goto loop2;
out2:
  i++;
  j = 0;
  goto loop1;
out1:
```

“For” Version:

?

Readability

- Data types and structures
 - Adequate predefined data types and structures
 - Example: Boolean vs. Integer
 - The presence of adequate facilities for defining data structures
 - Example: array of structs vs. collection of arrays (C)
- Syntactic design:
 - Identifier forms (allow long names).
 - Special words and methods of forming compound statements.
 - Form and meaning:
 - self-descriptive constructs, meaningful keywords.
 - Example (negative): static keyword in C has context dependent meaning.

Criteria for Language Evaluation

- **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 (e.g. program correctness).
- **Cost:** the ultimate total cost associated with a PL.

Writability

- **Simplicity and orthogonality**
 - Few constructs, a small number of primitives, a consistent, small set of rules for combining them (avoid misuse or disuse of features).
- **Support for abstraction**
 - The ability to define and use complex structures or operations in ways that allow details to be ignored
 - **Process Abstraction** (e.g. sorting algorithm implemented as a subprogram)
 - **Data Abstraction** (e.g. trees & lists in C++/Java vs. Fortran77).

Writability

- Expressivity
 - A set of relatively convenient ways of specifying operations.
 - Strength and number of operators and predefined functions.
 - Examples:
 - Increment operators in C.
 - Short circuit operators in Ada.
 - Counting loops with *for* vs. *while* in Java.

Criteria for Language Evaluation

- **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 (e.g. program correctness).
- **Cost:** the ultimate total cost associated with a PL.

Reliability

- Type checking
 - Testing for type errors at compile-time vs. run-time.
 - Examples:
 - (+) Ada, Java, C#, C++.
 - (–) C.
- Exception handling
 - Intercept run-time errors, take corrective measures and continue.
 - Examples:
 - (+) Ada, C++, Java.
 - (–) Fortran, C.

Reliability

- Aliasing
 - Presence of two or more distinct referencing methods for the same memory location:
 - Pointers in C, references in C++.
- Readability and writability
 - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches (less safe), and hence reduced reliability.

Criteria for Language Evaluation

- **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 (e.g. program correctness).
- **Cost:** the ultimate total cost associated with a PL.

Cost

- Training programmers to use the language.
- Writing programs (closeness to particular applications).
- Compiling programs:
 - Tradeoff between compile vs. execution time through optimization.
- Executing programs:
 - Example: many run-time type checks slow the execution (Java).
- Language implementation system:
 - Availability of free compilers.
- Reliability: poor reliability leads to high costs.
- Maintaining programs:
 - Corrections & adding new functionality.

Other Criteria

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

Outline

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

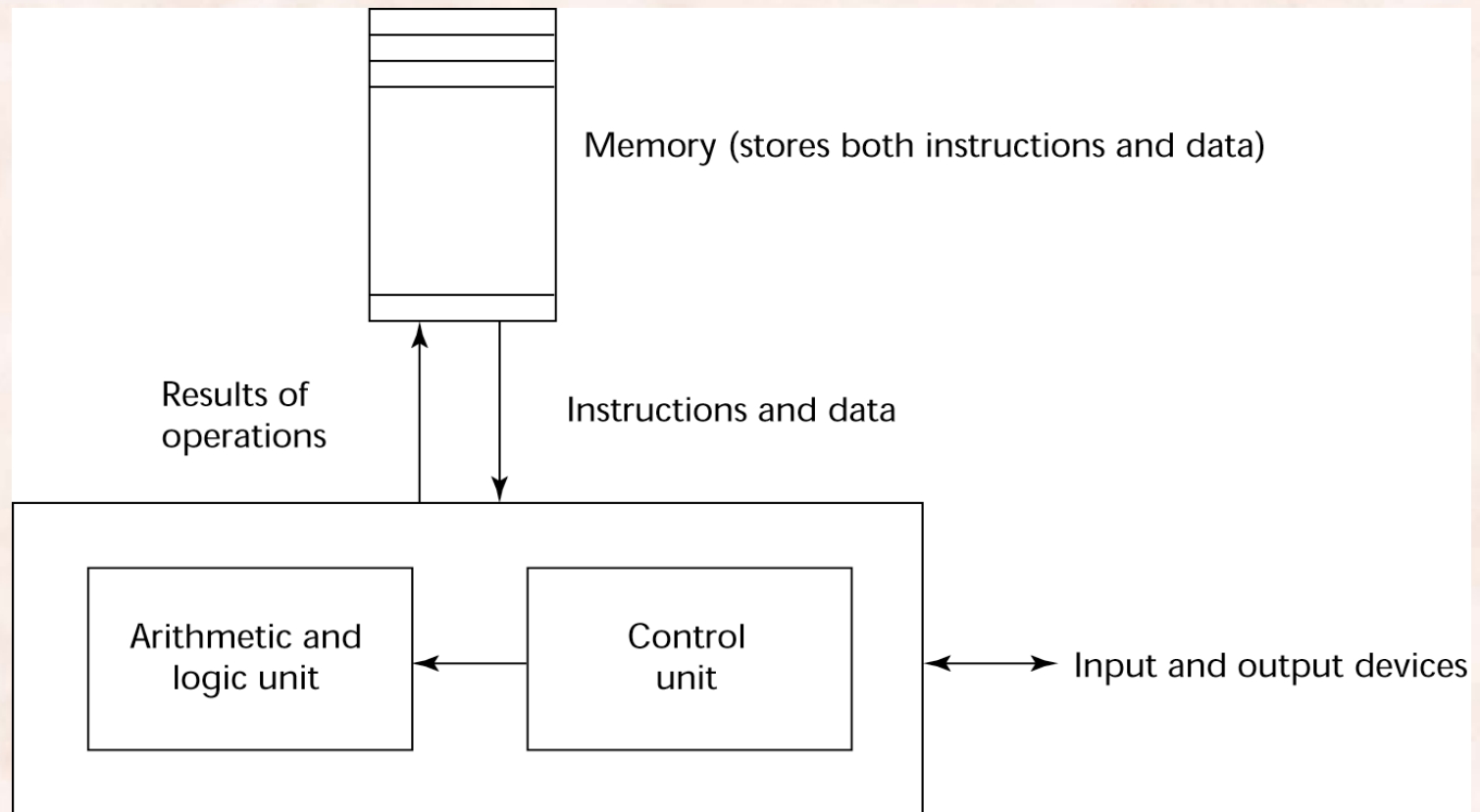
Influences on Language Design

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

Influences: Computer Architecture

- Most prevalent 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

Von Neumann Architecture



Central processing unit

Lecture 01

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
```

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

Influences: Programming Methodologies

- 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 design
 - abstract data types (Simula 67)
- Middle 1980s: Object-oriented programming
 - data abstraction + inheritance + polymorphism
 - Smalltalk, Ada 95, C++, Java, CLOS, Prolog++

Outline

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

Language Paradigms

- Imperative (Turing Machines – Alan Turing, 1912-1954)
 - Designed around the von Neumann architecture.
 - Computation is performed through statements that change a program's state.
 - Central features are variables, assignment statements, and iteration; sequencing of commands, explicit state update via assignment.
 - May include :
 - OO programming languages,
 - scripting languages,
 - visual languages.
 - Examples: Fortran, Algol, Pascal, C/C++, Java, Perl, JavaScript, Visual BASIC .NET

Language Paradigms

- **Functional (Lambda Calculus – Alonzo Church, 1903-1995)**
 - Main means of making computations is by applying functions to given parameters.
 - Examples: LISP, Scheme, ML, Haskell
 - May include OO concepts.
- **Logic (Predicate Calculus – Gotlob Frege, 1848-1925)**
 - Rule-based (rules are specified in no particular order).
 - Computations are made through a logical inference process.
 - Example: Prolog, CLIPS.
 - May include OO concepts.

Outline

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

Trade-offs in Language Design

- **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.

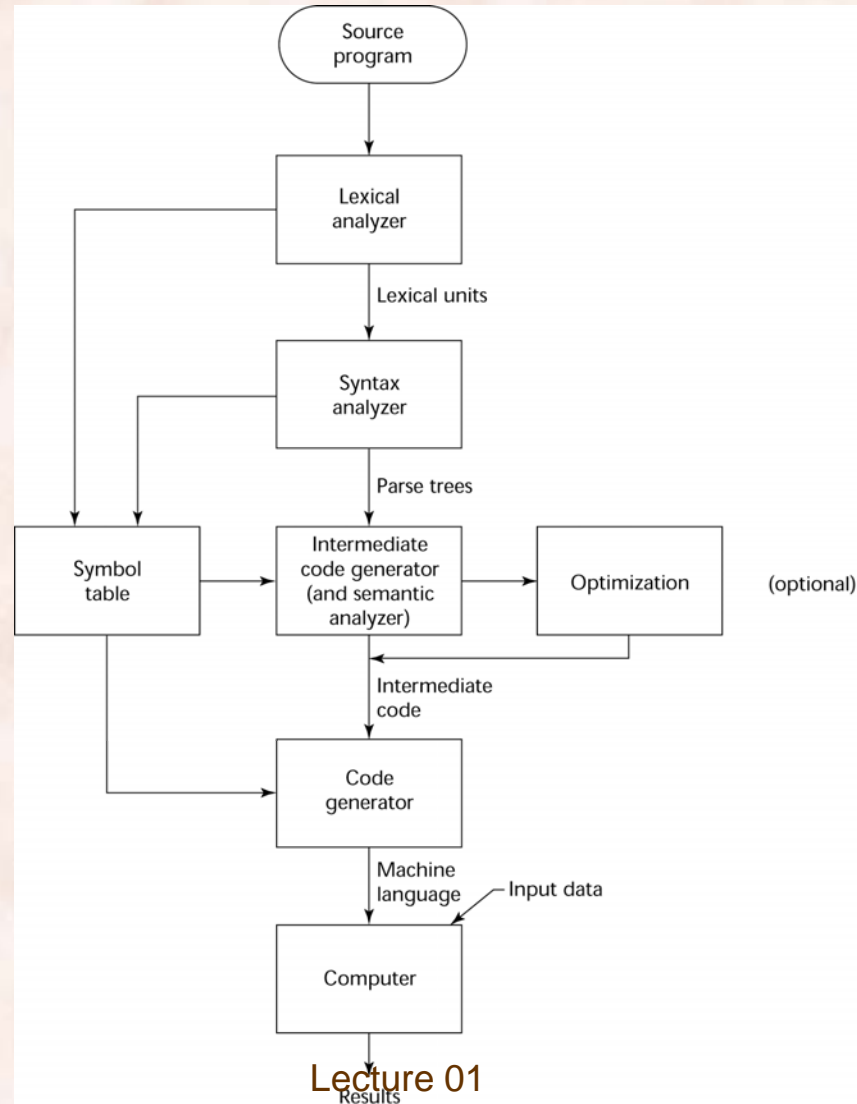
Implementation Methods

- **Compilation:**
 - Programs are translated into machine language & system calls.
- **Interpretation:**
 - Programs are interpreted by another program – an interpreter.
- **Hybrid:**
 - Programs are translated into an intermediate language that allows easy interpretation.
- **Just-in-Time:**
 - Hybrid + compile subprograms' code the first time they are called.

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 (e.g. identifiers, operators, keywords).
 - **syntactic analysis**: transforms lexical units into *parse trees* which represent the syntactic structure of program.
 - **semantics analysis**: check for errors hard to detect during syntactic analysis; generate *intermediate code*.
 - **code generation**: machine code is generated.

The Compilation Process



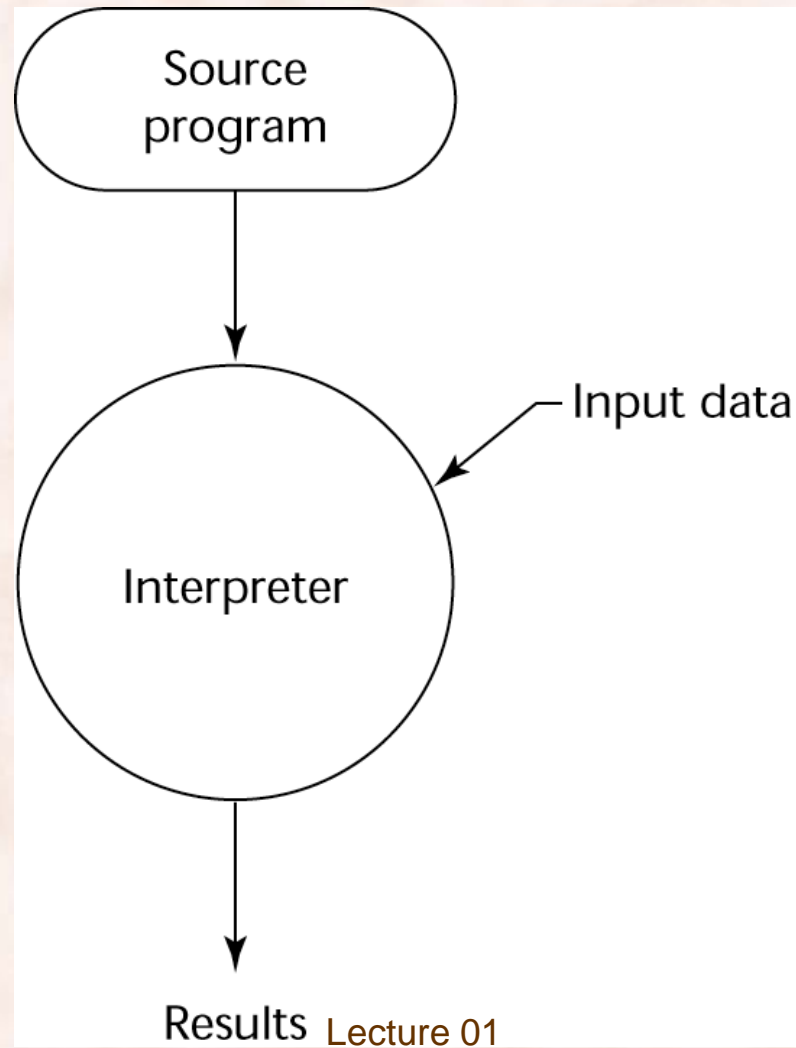
More Terminology

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

Interpretation

- Interpreter usually implemented as a *read-eval-print* loop:
 - read expression in the input language (usually translating it in some internal form)
 - evaluates the internal form of the expression
 - print the result of the evaluation
 - loops and reads the next input expression until exit.
- Interpreter acts as a virtual machine for the source language:
 - *fetch-execute* cycle replaced by the *read-eval-print* loop.
 - usually has a core component, called the interpreter “run-time”, that is a compiled program running on the native machine.

The Interpretation Process



Results Lecture 01

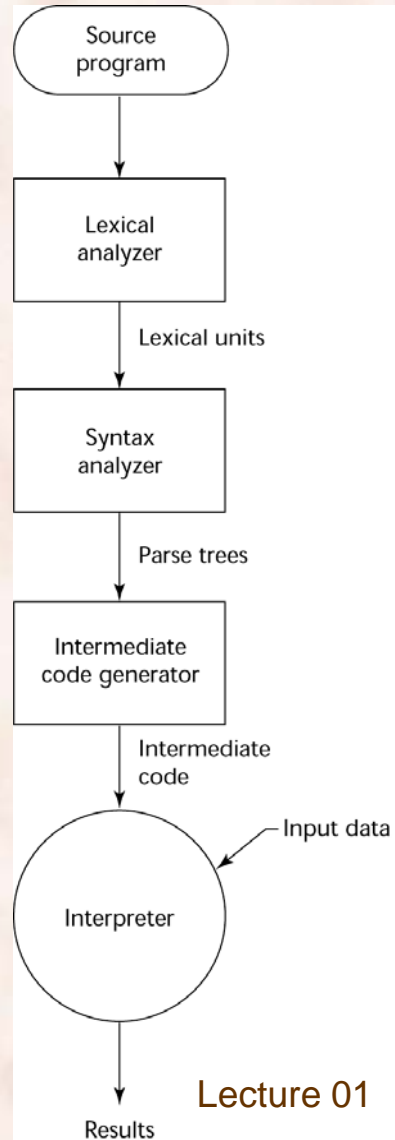
Interpretation

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

Hybrid Implementation

- 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*)

The Hybrid Implementation Process



Lecture 01

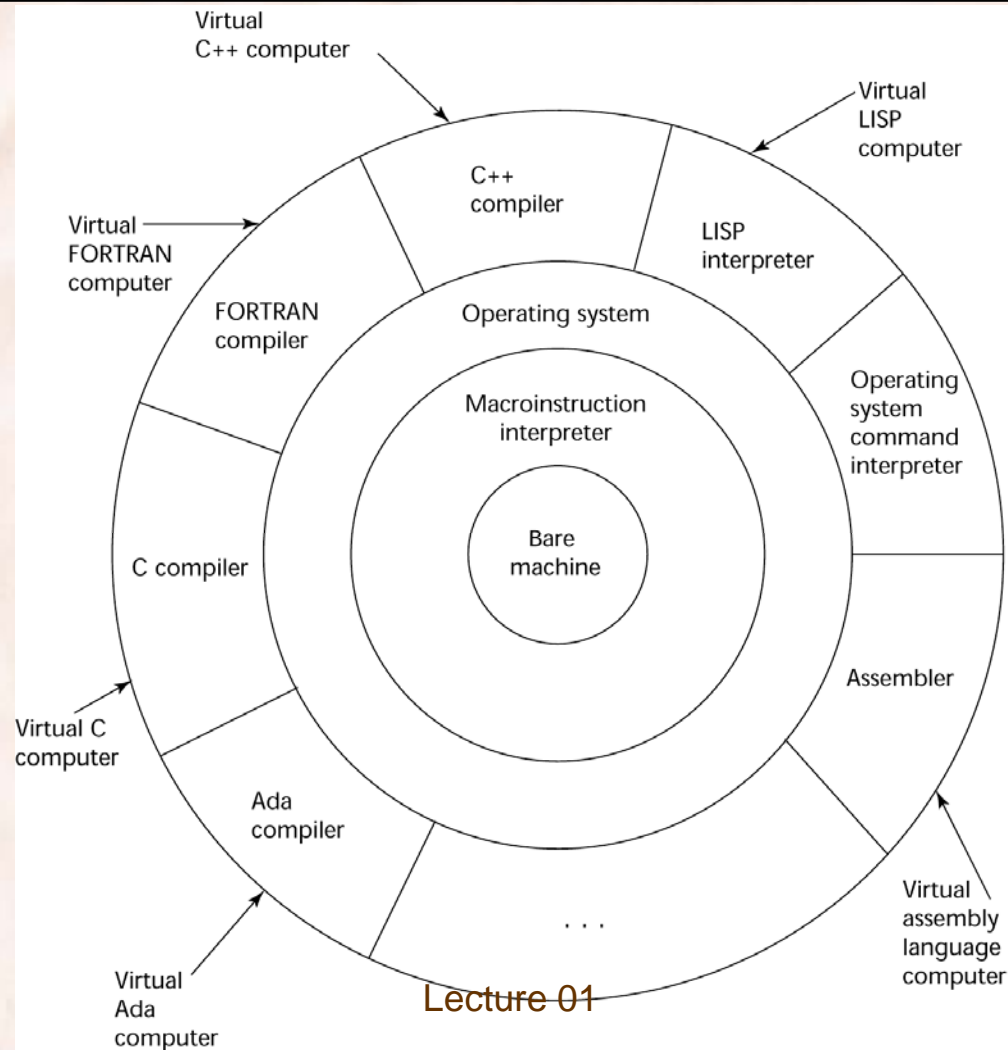
Just-in-Time Implementation

- 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.

Preprocessors

- Preprocessor is run before compiler, as a macro expander.
- Macros are commonly used to:
 - specify that code from another file is to be included;
 - define simple expressions/functions.
- A well-known example: C preprocessor
 - expands `#include`, `#define`, and similar macros.

Layered Interface of Virtual Computers



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, and cost.*
- Major influences on language design:
 - *machine architecture and software development methodologies.*
- Major implementation methods:
 - *compilation, pure interpretation, hybrid, and just-in-time.*