CS-3304 Introduction

In Text: Chapter 1

COURSE DESCRIPTION

2

What will you learn?

- Survey of programming paradigms, including representative languages
- Language definition and description methods
- Overview of features across all languages
- Implementation strategies

3

Semester Outline

- Introduction and Language Evaluation
- History and Evolution
- Syntax and Semantics
- Names, Typing, and Scoping
- Expressions and Assignment
- Control Structures
- Subprograms
- Functional & Declarative Languages
- Concurrency

4

Websites

- Course homepage: lecture notes and schedules

http://courses.cs.vt.edu/cs3304/Fall16/meng/

- Canvas website: lecture notes, assignments, grades, and announcements

https://canvas.vt.edu/courses/30688

5

INTRODUCTION

6

Overview

- Why are there so many programming languages?
- What makes a language successful?
- Why study programming languages?
- What types of programming languages are there?
- What are language implementation methods?
- What is the process of compilation?

Why are there so many PLs?

- Evolution: people have learned better ways of doing things over time
- Socio-economic factors: proprietary interests, commercial advantage
- Orientation towards special purposes
- Orientation towards special hardware
- Diverse ideas about what is pleasant to use

What makes a language successful?

- Expressive power (C, Algol-68, Perl)
  - Easy to express things
  - Although every language is Turing complete, language features have huge impact
  - We will focus on factors contributing to expressive power in the course
- Ease of use (Pascal, Java, Python)
  - Easy to learn

9

What makes a language successful?

- Ease of implementation (BASIC, Forth)
  - The languages can be implemented/installed on tiny machines
- Standardization (ANSI C)
  - To ensure portability of code cross platforms
- Open source (C)
  - With at least one open-source compiler or interpreter

10

What makes a language successful?

- Excellent compilers (Fortran, Common Lisp)
  - Possible to compile to very good (fast/ small) code
- Economics, Patronage, and Inertia
  - The backing of a powerful sponsor
  - E.g., COBOL and Ada by DoD, PL/1 by IBM

11

Why study PLs?

- 1. Make it easier to learn new languages
  - Some languages are similar; easy to walk down family tree
    - E.g., from Java to C#, from Pascal to C

12

Why study PLs?

- 2. Choose among alternative ways to express things based on the knowledge of implementation costs/performance overhead
  - Use simple arithmetic equivalents (use x*x instead of x^2)
  - Avoid call by value with large data items in Pascal

13

Why study PLs?

- 3. Simulate useful features in languages that lack them
  - Certain useful features are missing in some languages, but can be emulated by following a deliberate programming style
    - E.g., Older dialects of Fortran lack suitable control structures, so programmers can use comments and self-discipline to write well-structured code

14

Why study PLs?

- 4. Make better use of language technology whenever it appears
  - The code to parse, analyze, generate, optimize, and otherwise manipulate structured data can be found in almost any sophisticated program
  - Programmers with a strong grasp of the language technology will be able to write better structured and maintainable code

15

Why study PLs?

- 5. Get prepared to design new languages or extend existing languages
  - Easy-to-use
  - Easy-to-learn
  - Easy-code-to-maintain
  - … …

16

The PL spectrum

- Declarative
  - Functional — Lisp/Scheme, ML, Haskell
  - Dataflow — Id, Val
  - Logic, constraint-based — Prolog, SQL
- Imperative
  - von Neumann — C, Ada, Fortran
  - Object-oriented — Smalltalk, Eiffel, Java
  - Scripting — Perl, Python, PHP

17

Declarative vs. Imperative

- "High-level" vs. "Low-level"
- Programmers specify "what should be done" or "steps to do it"
- An example (C#): choose all odd numbers in a collection

```
var results =
collection.Where( num => num %
2 != 0);
```

```
List<int> results = new List<int>();
foreach(var num in collection)
{
    if (num % 2 != 0)
        results.Add(num);
}
```

18

Functional Languages

- Employ a computational model based on recursive definition of functions
- Take inspiration from the lambda calculus
  - A program is considered as a function from inputs to outputs, defined in terms of simpler functions through a process of refinements
- We will talk a lot about these languages

19

Dataflow Languages

- Model computation as the flow of information (tokens) among primitive functional nodes
- Provide an inherently parallel model:
  - Nodes are triggered by the arrival of input tokens, and can operate concurrently

20

Logic or Constraint-Based Languages

- Take inspiration from predicate logic
- Model computation as an attempt to find values that satisfy certain specified relationships, using goal-directed search through a list of logical rules

21

von Neumann Languages

- Most familiar and widely used
- The basic means of computation is the modification of variables

22

Object-oriented Languages

- Closely related to the von Neumann languages
- Have a much more structured and distributed model of both memory and computation
- Picture computation as interactions among semi-independent objects, each of which has both its own internal state and subroutines to manage that state

23

Scripting Languages

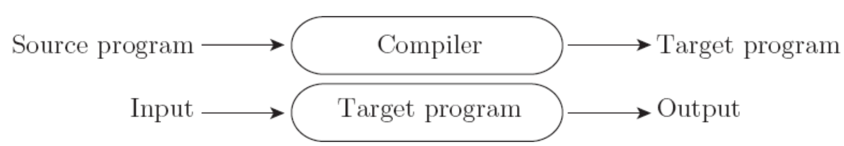
- Emphasize coordinating or "gluing together" components drawn from some surrounding context
- Support scripts, programs written for a special run-time environment that automate the execution of tasks, which could alternatively be executed one-by-one by a human creator

24

Language Implementation Methods

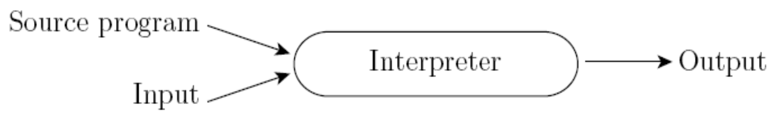
- Compilation
- Interpretation
- Hybrid

Compilation



- Translate high-level programs to machine code
- Slow translation
- Fast execution

Interpretation



- Interpret one statement and then execute it on a virtual machine
- No translation
- Slow execution
- E.g., Basic

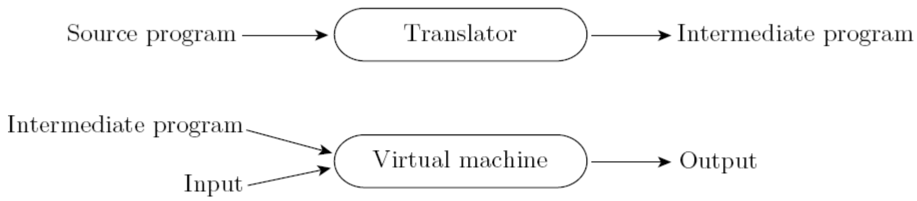
27

Compilation vs. Interpretation

- Compilation
  – Better performance
    - No runtime cost for interpretation
    - Program optimization
- Interpretation
  – Better diagnosis (with excellent source-level debugger)
  – Earlier diagnosis (execute erroneous program)

28

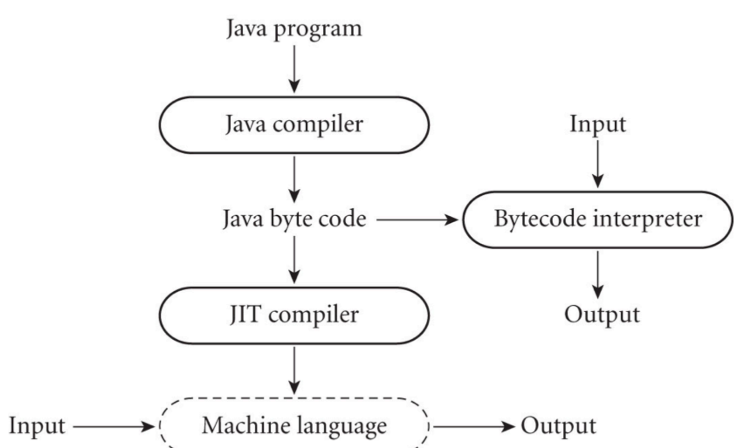
Hybrid Implementation



- Quick start in "Interpretation" mode
- Compile code on hot paths to speed up
  - E.g., Just-in-Time (JIT) compiler in Java Virtual Machine (JVM)
  - Dynamic profiling plays the trick

29

Hybrid Implementation (Java)



30

Implementation Strategies in Practice

- Preprocessing
- Library routines and linking
- Post-compilation assembly
- Source-to-source translation
- Bootstrapping

31

Preprocessing (Basic)

- An initial translator
  - to remove comments and white spaces,
  - to group characters together into tokens such as keywords, identifiers, numbers, and symbols,
  - to expand abbreviations in the style of a macro assembler, and
  - to identify higher-level syntactic structures, such as loops and subroutines
- Goal
  - To provide an intermediate form that mirrors the structure of the source, but can be interpreted more efficiently

32

Preprocessing (C)

- Conditional compilation
  - Delete portions of code to allow several versions of a program to be built from the same source