CS-3304 Introduction

In Text: Chapter 1 & 2

COURSE DESCRIPTION

What will you learn?

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

Semester Outline

• Introduction and Language Evaluation
• History and Evolution
• Syntax and Semantics
• Names, Typing, and Scoping
• Expressions and Assignment
• Control Structures
• Subprograms
• Functional Programming
• Logic Programming

Websites

• Course homepage: lecture notes and schedules
  http://courses.cs.vt.edu/cs3304/fall17/

• Canvas website: assignments, grades, and announcements
  https://canvas.vt.edu/courses/56037

INTRODUCTION TO PROGRAMMING LANGUAGES
Overview

• 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
• 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
• 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
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

• 2. 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

• 3. Choose among alternative ways to express things based on the knowledge of implementation costs/performance overhead
  • Use simple arithmetic equivalents (use \( x^2 \) instead of \( x \times x \))
  • Avoid call by value with large data items in Pascal
  • Manual vs. automatic memory management

• 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

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

A Story: ALGOL 60 vs. Fortran

• ALGOL 60 (Backus et al., 1963) was more elegant and had much better control statements than Fortran (McCracken, 1961)
• ALGOL 60 failed to displace Fortran
  – Poor understanding of the new language
  – No appreciation on the benefits of block structures, recursion, and various control structures
OVERVIEW OF PROGRAMMING LANGUAGES

Influences on Language Design
- Computer Architecture
- Programming Design Methodologies

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

1. Initialize the program counter
2. Repeat forever
   a. Fetch the instruction pointed by the counter
   b. Increment the counter
   c. Decode the instruction
   d. Execute the instruction

Programming Design Methodologies
- 1950s and early 1960s
  - Simple applications
  - Worry about machine efficiency and hardware cost
- Late 1960s: hardware costs decreased and programmer costs increased
  - Large and complex applications
  - People efficiency became important
  - Readability: better control structures
    - Structured programming
    - Top-down design and step-wise refinement
Programming Design Methodologies

- Late 1970s: Process-oriented to data-oriented
  - Data abstraction: using abstract data types
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

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

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

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

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

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

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
von Neumann Languages

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

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

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