CS-3304 Introduction

In Text: Chapter 1 & 2

INTRODUCTION TO PROGRAMMING LANGUAGES

Overview

- Why study programming languages?
- What types of programming languages are there?
- What are language implementation methods?
- What are Language design trade-offs?
- What is the process of compilation?

WHY STUDY PROGRAMMING LANGUAGES?

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?

- Easy to learn (BASIC, Pascal, LOGO, Scheme).
- Easy to express things, easy use once fluent, "powerful" (C, APL, Algol-68, Perl).
- Easy to implement (BASIC, Forth).
  - The languages can be implemented/installed on tiny machines
- Possible to compile to very good (fast/small) code (Fortran).
- Backing of a powerful sponsor (COBOL and Ada by DoD, PL/I by IBM).
- Wide dissemination at minimal cost (Pascal, Turing, Java).

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

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#

- 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*x instead of x^2)
  - 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

Reasons to Study Concepts of PLs

- Increased capacity to express programming concepts
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Understanding the significance of implementation
- Increased ability to design new languages
- Overall advancement of computing

LANGUAGE EVALUATION CRITERIA

Evaluating A Language

- 4 main criteria:
  - Readability
  - Writability
  - Reliability
  - Cost

Evaluation: Readability

- The most important criterion
- Overall simplicity
  - Too many features is bad
  - Multiplicity of features is bad
- Orthogonality
  - Combine the primitive elements to build the control and data structures
  - Meaning is context independent
  - E.g. data types
- Data types
  - Having a variety of them
  - e,.g. timeOut = 1, timeOut = true
- Syntax Design
  - Special/reserved words (while, for, class, …)

Evaluation: Writability

- Factors:
  - Simplicity and orthogonality
  - Expressivity
    - E.g. (in C) count++ is easier than count = count +1

Evaluation: Reliability

- Factors:
  - Type checking
    - Test for type errors
    - Compile-time type check is desirable, while run-time is expensive
  - Exception handling
    - How program intercepts unusual conditions/ run-time errors
    - E.g. floating-point overflow
  - Aliasing
    - Two or more references to the same memory
  - Readability and writability

Evaluation: Cost

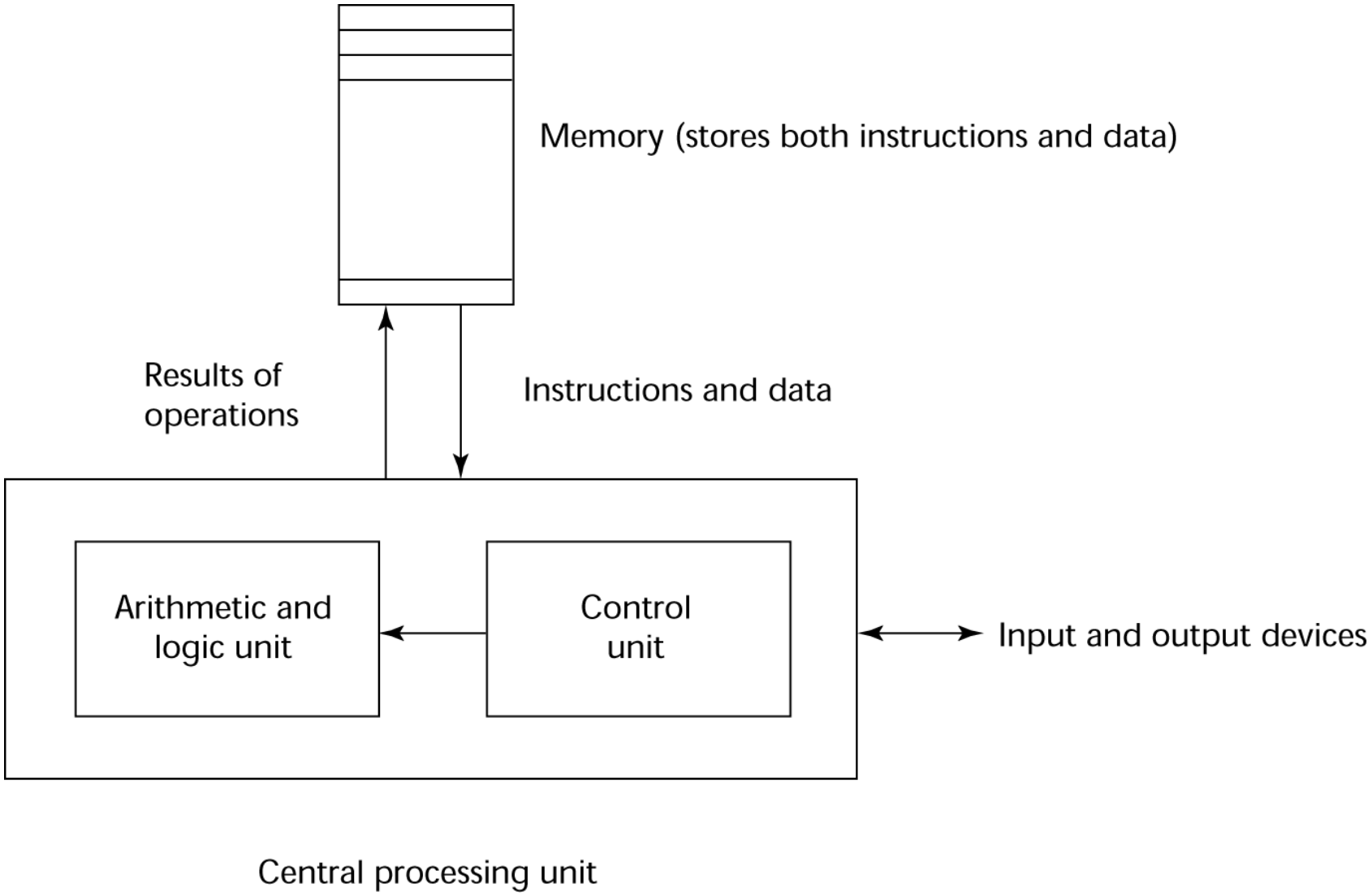
- Categories
  - Programmer training
  - Software creation (high-level L, lower cost)
  - Compiler cost
  - Execution (lots of type check)
  - Poor reliability
    - Failure cost could be very high: e.g. X-ray machine, cranes
  - Maintenance
    - Include correction and modification
    - Usually higher than development
- Other criteria: portability, generality, well-definedness

OVERVIEW OF PROGRAMMING LANGUAGES

Influences on Language Design

- Computer Architecture
- Programming Design Methodologies

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 Design Methodologies

- 1950s and early 1960s
  - Simple applications
  - Worry about machine efficiency and hardware cost

Programming Design Methodologies

- 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
  - Language deficiencies:
    - Incompleteness of type check
    - Inadequacy of control statements

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
  - Smalltalk

Language Categories

Programming paradigms:

- Procedural/Imperative
- Functional/Applicative
- Logic
- Object-oriented (closely related to imperative)
- Problem-oriented/application-specific

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

```
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);
}
```

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 Rule-Based Languages

- Take inspiration from predicate logic
- Rules are specified in no particular order
- Implementation chooses an order for rules to produce the desired result

Imperative Languages

- von Neumann Languages
- Most familiar and widely used
- The basic means of computation is the modification of variables
- The algorithm is identified in great detail, and specified order for instruction execution

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
- Inheritance ensures sw reuse

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
- Specify the layout of the information in web documents

Language Design Trade-offs

1. Reliability versus cost of execution
   - Java has array index range check
   - C does not
   - Result:
     - C programs execute faster
     - Java traded execution efficiency for reliability

Language Design Trade-offs

2. Writability versus readability
   - Example: APL
   - Has a lot of array operators
   - Is very writable
   - Can write huge amount of computation in a very small prg.
   - Has poor readability
   - Almost no one other than programmer can understand

Language Design Trade-offs

3. Writability versus reliability
   - C++: pointer can be manipulated in a variety of ways
   - Highly flexible in addressing data
   - Reliability problem
   - Java: does not have the feature