



CSE3302 PROGRAMMING LANGUAGES

CSE5307 PROGRAMMING LANGUAGE CONCEPTS

Kenny Q. Zhu
Dept. of Computer Science & Engineering
University of Texas at Arlington

KENNY Q. ZHU



Research Interests:

Natural Language Processing

Animal Language Processing
NLP for Mental Health
Knowledge representation/discovery

Programming Languages

Domain specific languages
Data Processing
Concurrency

Recent Publications:

AAAI, IJCAI, ACL, EMNLP,...

Degrees:

National University of Singapore (NUS)

Postdoc:

Princeton University

Experiences:

Microsoft Redmond

Microsoft Research Asia

Shanghai Jiao Tong University

Joined *UT Arlington* in fall 2023

ADMINISTRATIVE INFO (I)

- Hybrid course (both undergrad & graduate)
- Lecturer:
 - Kenny Zhu, ERB-535, kenny.zhu@uta.edu
 - Office hours: Wed 1-2 PM, ERB 535
- Teaching Assistant:
 - Peter Zhang, ERB-316, wxz9630@mavs.uta.edu
 - Office hours: Fri 2-4 PM
- Course Web Page (definitive source!):
<https://kenzhu2000.github.io/cse3302/>
- Materials may be optionally uploaded to Canvas as well

ADMINISTRATIVE INFO (II)

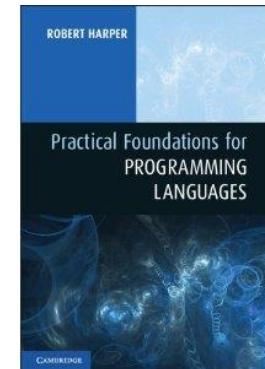
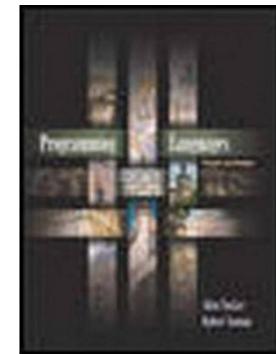
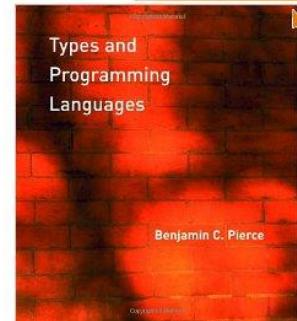
- Format:

- 1.5 hour lecture on Monday
- 0.5 hour lecture and 1 hour tutorial discussion on Wednesday
- Tutorials are led by TA

- Reference Texts:

- **Types and Programming Languages** by Benjamin C. Pierce, The MIT Press.
- **Programming Languages – Principles and Paradigms**, 2nd Edition, by Tucker & Noonan, McGraw Hill
- **Practical Foundations for Programming Languages** by Robert Harper, Cambridge University Press

- Lecture materials on course web page



ADMINISTRATIVE INFO (III)

- 3-credit course (16 weeks)
- Modes of Assessment:
 - In-class quizzes: 15%
 - Tutorial discussion participation: 5% (bonus)
 - Assignments: 20%
 - Programming Project: 20%
 - Final Exam: 45%
- Quizzes
 - Given out at random times
 - Usually on-screen **multiple-choice questions or short answer questions**
 - Bring piece of paper and a pen every time!
 - Submit answer after class (immediately) to TA or me
- Tutorials
 - Discuss assignment questions, issues in project, other Q&A
 - You will be asked to present your solutions
 - Volunteer to win tutorial participation points

ADMINISTRATIVE INFO (IV)

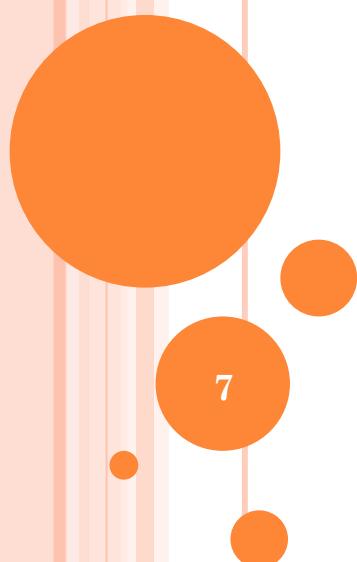
- Assignments

- Released (usually) every Wednesday)
- Due date printed on assignment sheet
- Submit solutions including code and data on Canvas
- Late submission: 50% off for 1 day, 0 afterwards
- Assignment solutions to be discussed at the tutorial in the following week (led by TA)

- Programming Project

- Individual project
- Implement an interpreter for a simple language called simPL
- Be able to run test programs and produce correct evaluation results
- Produce a report + code + results: due end of semester

INTRODUCTION



WHY DO WE LEARN PROGRAMMING LANGUAGES?

TWO MISCONCEPTIONS ABOUT THIS COURSE

- “This course about **programming**.”
- “This is another **compiler** course.”

Programming is about mastering the use of a language.

Compiler is about implementing a system that can parse a program in a high-level language into an intermediate form and then generate machine code. The focus is practical issues such as time and space complexity, code redundancy, and optimization.

WHAT THIS COURSE IS ABOUT

- *Theoretical aspects* of the **design** and **implementation** of **all** programming languages.
- The **commonalities** and **differences** between various *paradigms* and *languages*.
- So that you can:
 - Pick the right language for a project;
 - Design your own language (features);
 - Do programming language research.

OUTLINE OF TODAY'S LECTURE

- Principles
- Paradigms
- Special Topics
- A Brief History
- On Language Design
- Compilers and Virtual Machines
- Roadmap of This Course

THE FACTORIAL PROGRAM

- $n! = 1 * 2 * \dots * n$
 - Or $n! = \prod_{i=1}^n i$
- Two different mathematical languages

In computing, there are many more ways to do this ...

THE FACTORIAL PROGRAM

C:

```
int factorial(int n) {  
    int x = 1;  
    while (n>1) {  
        x = x * n;  
        n = n -1;  
    }  
    return x;  
}
```

Java:

```
class Factorial  
{  
    public static int fact(int n) {  
        int c, fact = 1;  
        if ( n < 0 )  
            System.out.println("Wrong Input!");  
        else {  
            for ( c = 1 ; c <= n ; c++ )  
                fact = fact*c;  
            return fact;  
        }  
    }  
}
```

THE FACTORIAL PROGRAM

Scheme:

```
(define (factorial n)
  (if (< n 1) 1
    (* n (factorial (- n 1))))
))
```

Prolog:

```
factorial(0, 1).
factorial(N, Result) :-
    N > 0, M is N - 1,
    factorial(M, SubRes),
    Result is N * SubRes.
```

PRINCIPLES

Programming languages have four properties:

- Syntax
- Names
- Types
- Semantics

For any language:

- Its designers must define these properties
- Its programmers must master these properties

SYNTAX

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

When studying syntax, we ask questions like:

- What is the basic vocabulary?
- What is the grammar for the language?
- How are syntax errors detected?

SYNTAX

```
class Factorial
{
    public static int fact(int n) {
        int c, fact = 1;
        if ( n < 0 )
            System.out.println("Wrong Input!");
        else {
            for ( c = 1 ; c <= n ; c++ )
                fact = fact*c;
            return fact;
        }
    }
}
```

Vocabulary of
Tokens:

Literal (constant)
Identifier
Operator
Separator (punctuation)
Reserved keyword

NAMES (IDENTIFIERS)

Various kinds of entities in a program have names:

variables, types, functions, parameters, classes, objects, ...

An entity is **bound** to a name (identifier) within the context of:

- Scope (static/dynamic)
- Visibility (part of scope that is visible)
- Lifetime (dynamic and runtime)
- Type

NAMES (IDENTIFIERS)

```
class Factorial
{
    public static int fact(int n) {
        int c, fact = 1;
        if ( n < 0 )
            System.out.println("Wrong Input!");
        else {
            for ( c = 1 ; c <= n ; c++ )
                fact = fact*c;
            return fact;
        }
    }
}
```

TYPES

A ***type*** is a collection of values and a collection of all *permissible* operations on those values.

- Simple types
 - numbers, characters, booleans, ...
- Structured types
 - Strings, lists, trees, hash tables, ...
- Function types
 - Simple operations like +, -, *, /
 - More complex/general function: $\text{int} \rightarrow \text{int}$
- Generic types (polymorphism): α
- A language's *type system* can help:
 - Determine permissible (legal) operations
 - Detect type errors

TYPES

```
class Factorial {  
    public static int fact(int n) {  
        int c, fact = 1;  
        if ( n < 0 )  
            System.out.println("Wrong Input!");  
        else {  
            for ( c = 1 ; c <= n ; c++ )  
                fact = fact*c;  
            return fact;  
        }  
    }  
}
```

int → int

SEMANTICS

The meaning of a program is called its *semantics*.

In studying semantics, we ask questions like:

- When a program is running, what happens to the values of the variables? (**operational semantics**)
- What does each expression/statement mean? (**static semantics**)
- What underlying model governs run-time behaviors, such as function calls? (**dynamic semantics**)
- How are objects allocated to memory at run-time?

SEMANTICS

```
class Factorial
{
    public static int fact(int n) {
        int c, fact = 1;
        if ( n < 0 )           ← Static Semantics
            System.out.println("Wrong Input!");
        else {
            for ( c = 1 ; c <= n ; c++ )
                fact = fact*c;   ← Operational Semantics
            return fact;
        }
    }
}
```

The diagram illustrates the execution flow of the Factorial class. It shows the flow from the 'if' condition to the 'Static Semantics' label, and from the assignment statement to the 'Operational Semantics' label. It also shows the flow from the 'return fact;' statement to the 'value' box, indicating the return value of the method. The 'reference' box points to the opening brace of the method definition, indicating the current scope.

PARADIGMS

- A programming *paradigm* is a pattern of problem-solving thought that underlies a particular *genre* of programs and languages.

a category of artistic composition, as in music or literature, characterized by similarities in form, style, or subject matter.

- There are four main programming paradigms:
 - Imperative
 - Object-oriented
 - Functional
 - Logic (declarative)

IMPERATIVE PARADIGM

- Follows the classic von Neumann-Eckert model:
 - Program and data are indistinguishable in memory
 - Program = a sequence of commands
 - State = values of all variables when program runs
 - Large programs use procedural abstraction
- Example imperative languages:
 - Cobol, Fortran, C, Ada, Perl, ...

THE VON NEUMANN-ECKERT MODEL

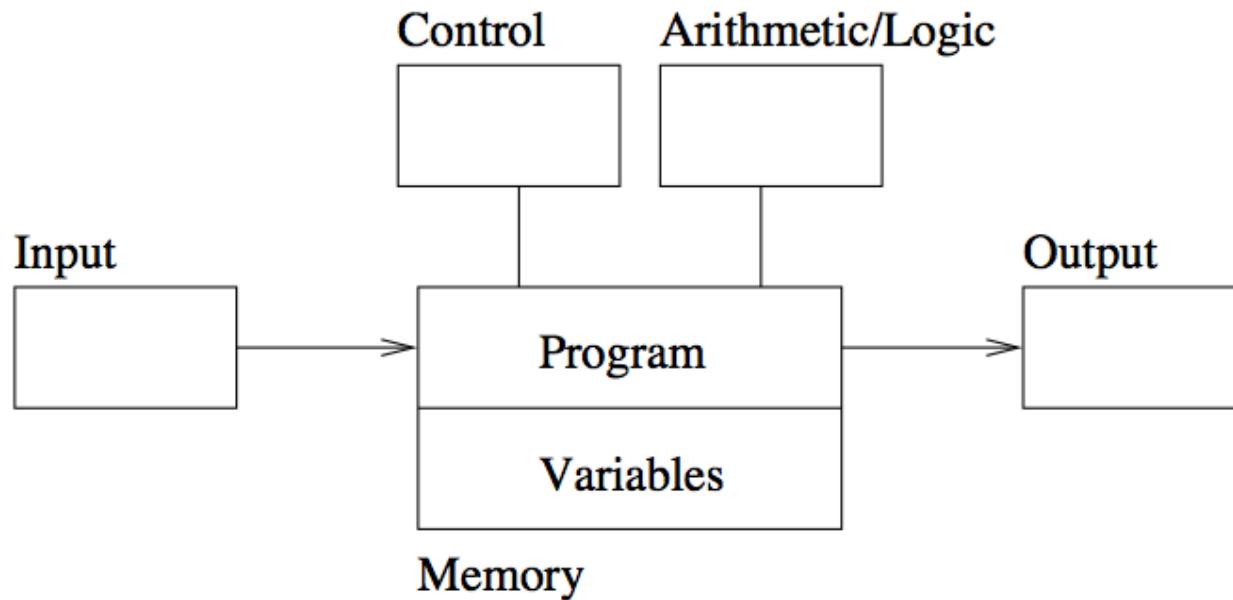


Figure 1.1: The von Neumann-Eckert Computer Model

OBJECT-ORIENTED (OO) PARADIGM

- An OO Program is a collection of objects that interact by passing messages that transform the state.
- When studying OO, we learn about:
 - Sending Messages → objects are active
 - Inheritance
 - Polymorphism
- Example OO languages:
 - *Smalltalk, Java, C++, C#, and Python*

FUNCTIONAL PARADIGM

- Functional programming models a computation as a collection of mathematical functions.
 - Set of all inputs = domain
 - Set of all outputs = range
- Functional languages are characterized by:
 - Functional composition
 - Recursion
 - No state changes: no variable assignments
 - $x := x + 1$ (wrong!)
 - Mathematically: output results instantly
- Example functional languages:
 - Lisp, Scheme, ML, Haskell, ...

LOGIC PARADIGM

- Logic programming declares *what* outcome the program should accomplish, rather than *how* it should be accomplished.

parent(X, Y) :- father(X, Y).

parent(X, Y) :- mother(X, Y).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

?- grandparent(X, jim).

- Declarative!

- When studying logic programming we see:

- Programs as sets of constraints on a problem
- Programs that achieve all possible solutions
- Programs that are nondeterministic

- Example logic programming languages:

- Prolog, CLP

MODERN LANGUAGES ARE MULTI-PARADIGM

- Haskell (F + I)
- Scala (F + I + O)
- OCaml (F + I + O)
- F Sharp (F + I + O)
- Python (O + I + F)
- ...

SPECIAL TOPICS

- Concurrency
 - E.g., Client-server programs
- Event handling
 - E.g., GUIs, home security systems
- Correctness
 - How can we prove that a program does what it is supposed to do under all circumstances?
 - “Program verification”
 - Why is this important???

A BRIEF HISTORY

How and when did programming languages evolve?
What communities have developed and used them?

- Artificial Intelligence – Prolog, CLP, Python
- Computer Science Education – Pascal, Logo
- Science and Engineering – Fortran, Ada, ML, Haskell
- Information Systems – Cobol, SQL
- Systems and Networks – C, C++, Perl, Python
- World Wide Web – HTML, Java, Javascript, PHP

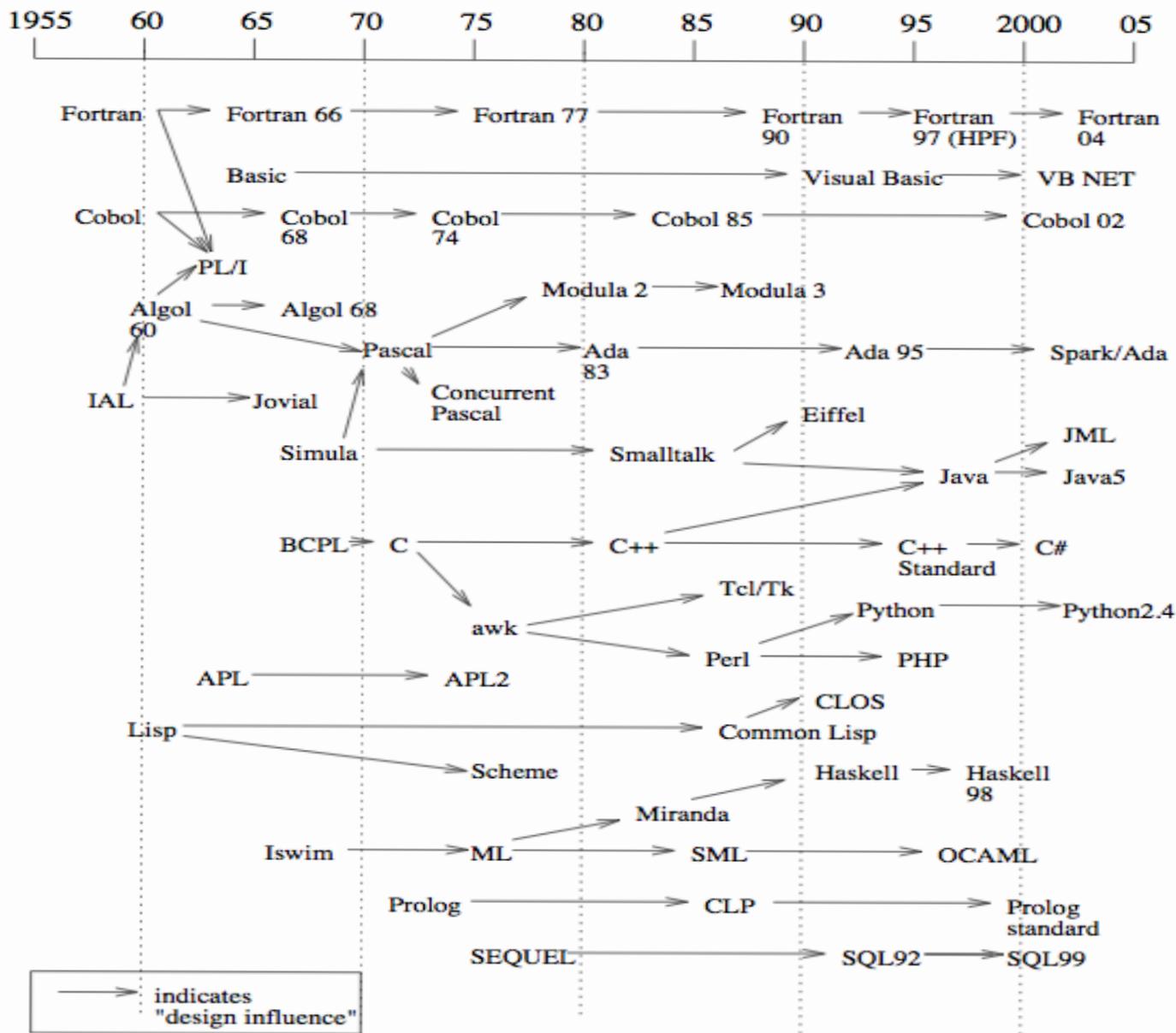


Figure 1.2: A Snapshot of Programming Language History

70 YEARS OF PROGRAMMING LANGUAGES HISTORY IN 8 MINS

“The most popular programming languages 1955-2025”

https://www.youtube.com/watch?v=5yAbVkIMl_M