CS383 Programming Languages

Kenny Q. Zhu
Dept. of Computer Science
Shanghai Jiao Tong University
Kenny Q. Zhu

Research Interests:

Programming Languages
Data Processing
Concurrency

Data & Knowledge Engineering
Information extraction
Knowledge discovery
Natural language understanding

Recent Publications:
TACL 2014, ICDE2013, SIGMOD’12, KDD’12, SIGCOMM’11, ICFP’11, PADL 2010, POPL 2008, ...

Degrees: National University of Singapore (NUS)
Postdoc: Princeton University
Experiences: Microsoft Redmond, USA
Microsoft Research Asia
Research Professor (PhD. Advisor) at SJTU since 2009
Director of ADAPT Lab
ADMINISTRATIVE INFO (I)

- All-English Course: everything in English!
- Lecturer:
  - Kenny Zhu, SEIEE #03-541, kzhu@cs.sjtu.edu.cn
  - Office hours: by appointment or after class
- Teaching Assistant:
  - Yu Gong, SEIEE #03-341, gy910210@163.com
  - Office hours: Thursday 16:00 - 17:00
- Course Web Page (definitive source!):
ADMINISTRATIVE INFO (II)

- Format:
  - Lectures on Mon (2 periods) & Wed (2 periods)
  - Tutorials on Fri (2 periods) – Led by Yu; Your participation is REQUIRED!

- Reference Texts:
  - Practical Foundations for Programming Languages by Robert Harper, Cambridge University Press
  - Types and Programming Languages by Benjamin C. Pierce, The MIT Press.

- Lecture materials on course web page
ADMINISTRATIVE INFO (III)

- 3-credit course
- Modes of Assessment:
  - In-class quizzes: 10%
  - Tutorial participation: 5%
  - Assignments: 30%
  - Programming Project: 25%
  - Final Exam: 30%
- Quizzes
  - Given out at random times
  - 10 on-screen multiple choice questions
  - Bring piece of paper and a pen every time!
- Tutorials
  - Typically on Friday
  - Discuss assignment questions and also issues in project
  - You will be asked to present your answers
  - Volunteer to win extra scores!
Administrative Info (IV)

- Assignments
  - Released (usually) on Wednesday
  - Due on the following Wednesday
  - Submit hard copies to me during Wednesday’s class or to Yu’s office
  - Submit code/data to Yu’s email
  - Late submission: -30% of full score for each additional day
  - Assignment solutions to be discussed at the tutorial on Friday following the submission

- 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
Why Do we Learn Programming Languages?
TWO MYTHS ABOUT THIS COURSE

“This course about programming.”

“This is another compiler course.”
What This Course is About

- *Theoretical aspects* of the design and implementation of programming languages.

- The **commonalities** and **differences** between various *paradigms* and *languages*.

So you can:
- Pick the right language for a project;
- Design your own language (features);
- Do programming language research.
OUTLINE

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

- $n! = 1 \times 2 \times \ldots \times n$

- Or $n! = \prod_{i=1}^{n} i$

THE FACTORIAL PROGRAM

C:

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

Java:

```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.
```
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?
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;
        }
    }
}

Various kinds of entities in a program have names: variables, types, functions, parameters, classes, objects, ...

Named entities are bound in a running program to:

- Scope
- Visibility
- Type
- Lifetime
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 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: \( \rightarrow \) (arrow) type

- A language’s *type system* can help to:
  - Determine legal operations
  - Detect type errors
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 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 behavior, such as function call? (dynamic semantics)
- How are objects allocated to memory at run-time?
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;
        }
    }
}
**Paradigms**

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

- 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!)

- 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).
  
  - 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
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?
  - 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
- 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
ON LANGUAGE DESIGN

Design Constraints

- Computer architecture
- Technical setting
- Standards
- Legacy systems

Design Outcomes and Goals
Figure 1.3: Levels of Abstraction in Computing

- Natural language
- Application area
- Programming language
- Compiler/interpreter
- User interface (IDE)
- Operating system
- Machine language
WHAT MAKES A SUCCESSFUL LANGUAGE?

Key characteristics:
- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation
Simplicity and Readability

- Small instruction set
  - E.g., Java vs. Scheme

- Simple syntax
  - E.g., C/C++/Java vs. Python

- Benefits:
  - Ease of learning
  - Ease of programming
A language element is bound to a property at the time that property is defined for it.

So a *binding* is the association between an object and a property of that object.

- Examples:
  - a variable and its type
  - a variable and its value

- Early binding takes place at compile-time
- Late binding takes place at run time
A language is *reliable* if:

- Program behaviour is the same on different platforms
  - E.g., early versions of Fortran
- Type errors are detected
  - E.g., C vs. Haskell
- Semantic errors are properly trapped
  - E.g., C vs. C++
- Memory leaks are prevented
  - E.g., C vs. Java
LANGUAGE SUPPORT

- Accessible (public domain) compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)
  - Eclipse vs. vim
  - Visual Studio vs. Emacs
ABSTRACTION IN PROGRAMMING

- **Data**
  - Programmer-defined types/classes
  - Class libraries

- **Procedural**
  - Programmer-defined functions
  - Standard function libraries
Orthogonality

A language is *orthogonal* if its features are built upon a small, *mutually independent* set of primitive operations.

- **while** loop vs. **for** loop in C

Fewer exceptional rules = conceptual simplicity

- E.g. our tutorials are “usually” on Friday except the last week of each month or when Yu is busy with his research on action conceptualization …
- E.g., restricting types of arguments to a function

Tradeoffs with efficiency
EFFICIENT IMPLEMENTATION

- Embedded systems
  - Real-time responsiveness (e.g., navigation)
  - Failures of early Ada implementations
- Web applications
  - Responsiveness to users (e.g., Google search)
- Corporate database applications
  - Efficient search and updating
- AI applications
  - Modeling human behaviors
COMPILERS AND INTERPRETERS

- Compiler – produces machine code
- Interpreter – executes instructions on a virtual machine
- Example compiled languages:
  - Fortran, Cobol, C, C++
- Example interpreted languages:
  - Scheme, Haskell, Python, Perl
- Hybrid compilation/interpretation
  - The Java Virtual Machine (JVM)
    - .java → .class
    - .class executes on JVM
THE COMPILING PROCESS

Figure 1.4: The Compile-and-Run Process
THE INTERPRETING PROCESS

Figure 1.5: Virtual Machines and Interpreters
Course Roadmap

- Mathematic foundation – inductive definition and inductive proofs
- Untyped Lambda Calculus
- Simply-typed Lambda Calculus
- Extensions to Simply-typed Lambda Calculus
- Going Imperative
- Memory Management
- Subtyping
- Type Inference

- Case Study: Imperative Programming
- Case Study: Object-Oriented Programming
- Case Study: Functional Programming
- Case Study: Logic Programming
- Special Topic: Concurrency