CS383 Programming Languages

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

Kenny Q. Zhu

Research Interests:

Artificial Intelligence

- Knowledge representation/discovery
- Natural language understanding
- Natural language generation

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, USA*
	Microsoft Research Asia
	Faculty at *SJTU* since 2009
	Director of **ADAPT Lab**

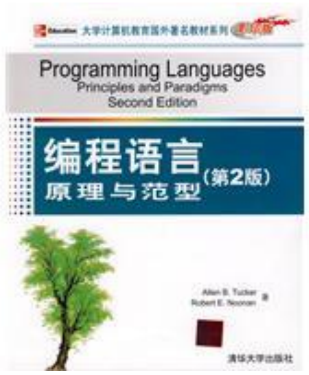
Administrative Info (I)

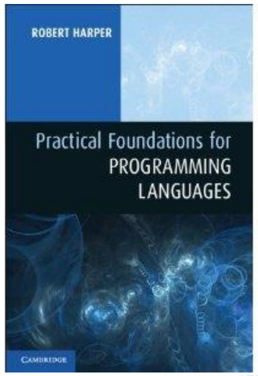
- All-English Course: everything in English!
- Lecturer:
  - Kenny Zhu, SEIEE #03-407, kzhu@cs.sjtu.edu.cn
  - Office hours: by appointment or after class
- Teaching Assistant:
  - Bran Li, SEIEE #03-329, likaijian@sjtu.edu.cn
  - Yvonne Huang, SEIEE #03-341, Yvonne_huang@sjtu.edu.cn
  - Office hours: Thursday 16:00 - 17:00
- Course Web Page (definitive source!): http://www.cs.sjtu.edu.cn/~kzhu/cs383/

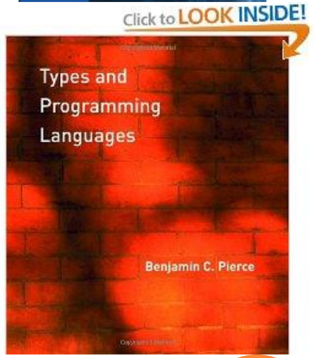
Administrative Info (II)

- Format:
  - Two lecture classes on Monday
  - Followed by a tutorial on Monday – Led by TA; Your participation is REQUIRED!

- 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 / Tsinghua University Press
  - **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:	10%
Tutorial participation:	5%
Assignments:	30%
Programming Project:	25%
Final Exam:	30%

- Quizzes
  - Given out at random times
  - Usually on-screen **multiple choice questions**
  - Bring piece of paper and a pen every time!
  - Submit answer after class (immediately) to TA
- Tutorials
  - Typically after every two lectures
  - Discuss assignment questions, issues in project, other Q&A
  - You will be asked to present your answers
  - Volunteer to win extra scores!

Administrative Info (IV)

- Assignments
  - Released (usually) every week (two lectures)
  - Due date printed on assignment sheet
  - Submit solutions including code and data on Canvas
  - Late submission: -30% of full score for each additional day
  - Assignment solutions to be discussed at the tutorial following the submission (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

Wechat Group

CS383-2022spring

该二维码7天内(2月19日前)有效，重新进入将更新

Introduction

Why Do we Learn Programming Languages?

Two Misconceptions 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 all 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 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 * … * 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

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

```
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 legal 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: int → int
- Generic types (polymorphism): α
- A language's *type system* can help:
  - Determine 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 behavior, such as function call? (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 )
       System.out.println(“Wrong Input!");
      else {
         for ( c = 1 ; c <= n ; c++ )
            fact = fact*c;
         return fact;
      }
   }
```

Static Semantics

Operational Semantics

reference

value

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

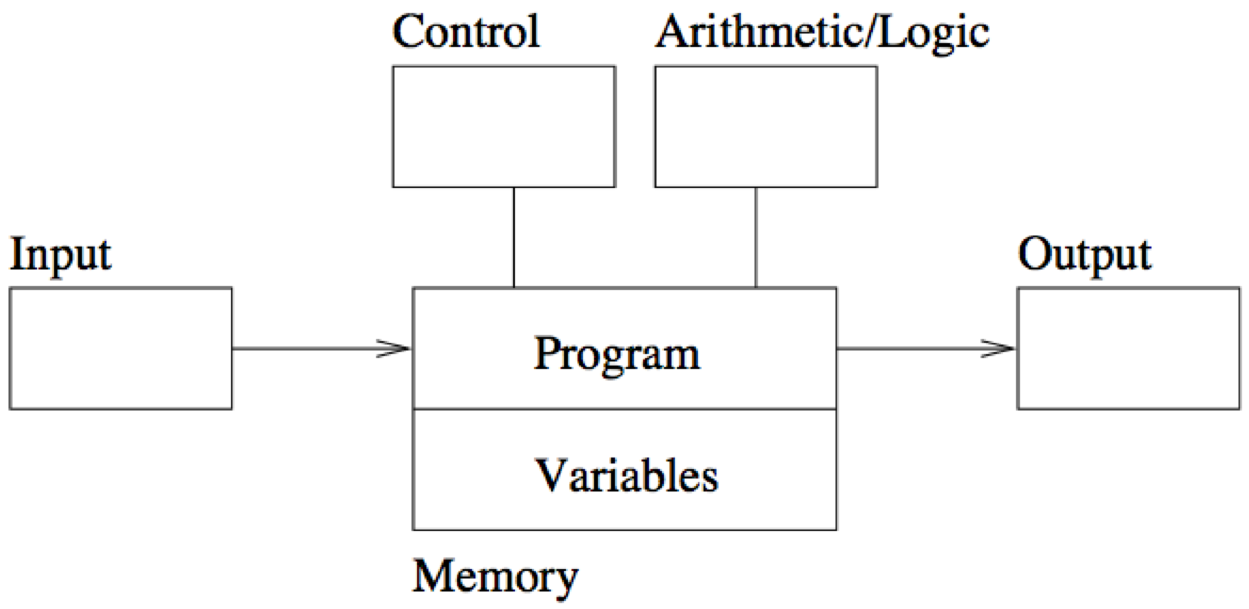


Figure 1.1: The von Neumann-Eckert Computer Model

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?
  - 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

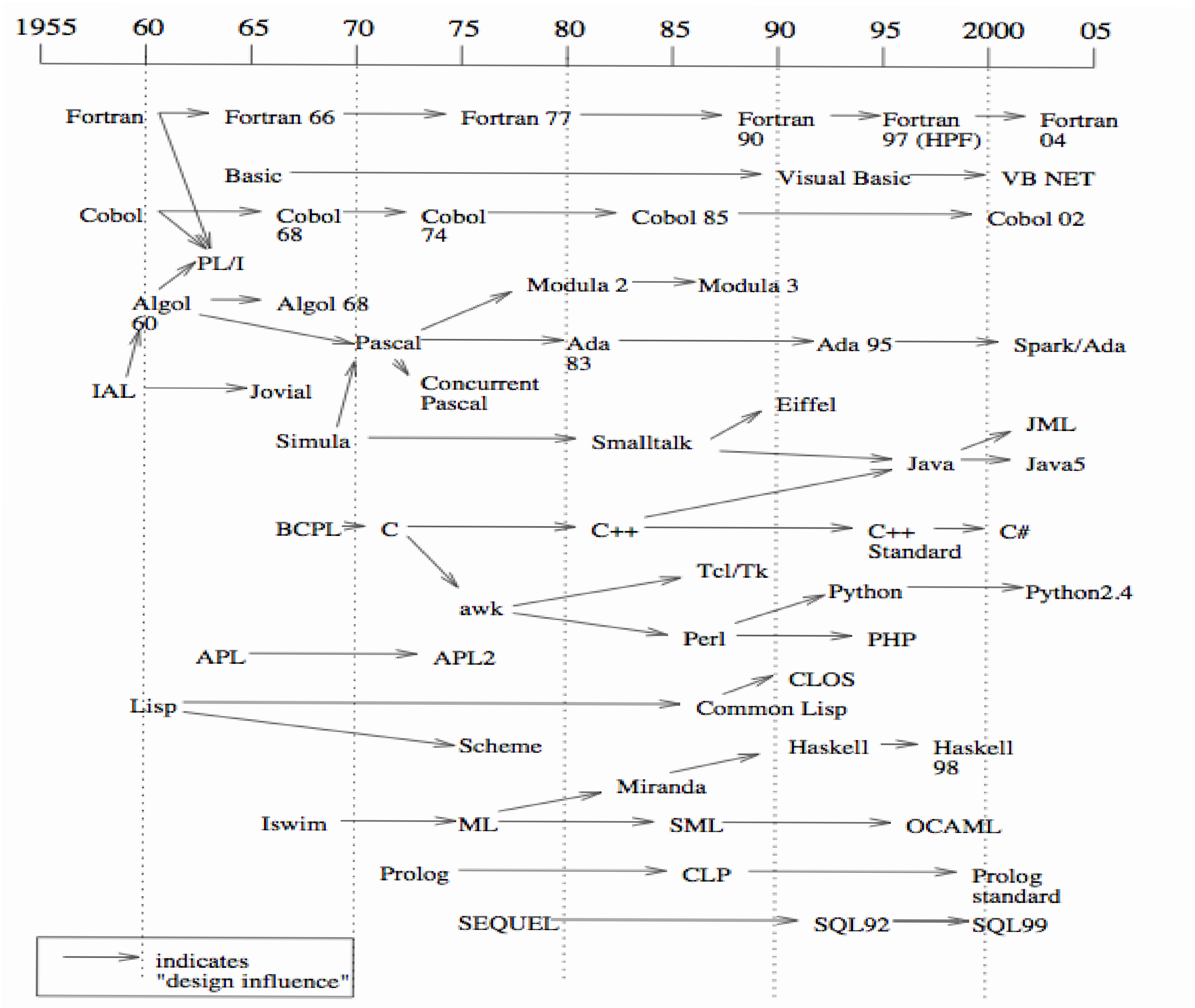


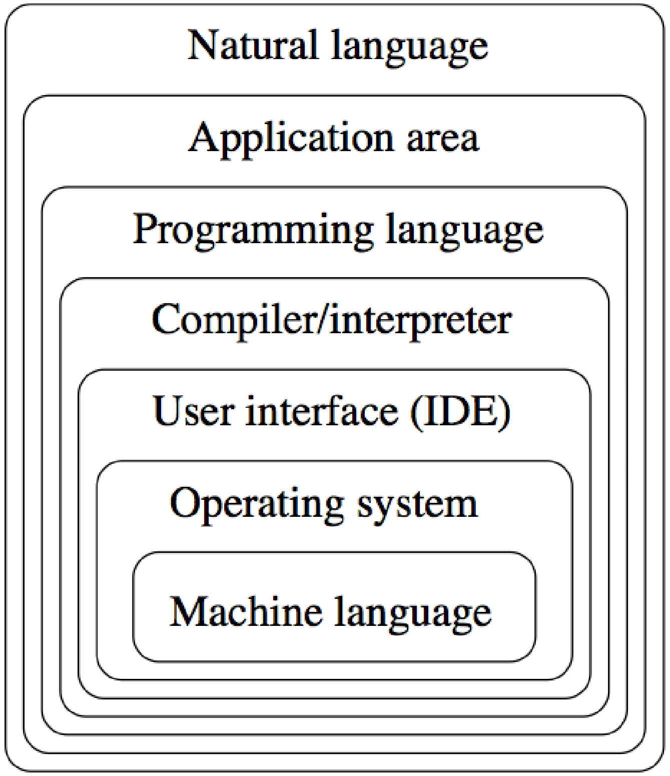
Figure 1.2: A Snapshot of Programming Language History

On Language Design

Design Constraints

- Computer architecture
- Technical setting
- Standards
- Legacy systems

Design Outcomes and Goals



Levels of abstraction in computing

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

Clarity about Binding

- 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

Reliability

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
  - Java (open) vs. C# (closed)
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)
  - Jupyter Notebook 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 Monday except the last week of each month or when the TA is busy with his research on text generation…
  - 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

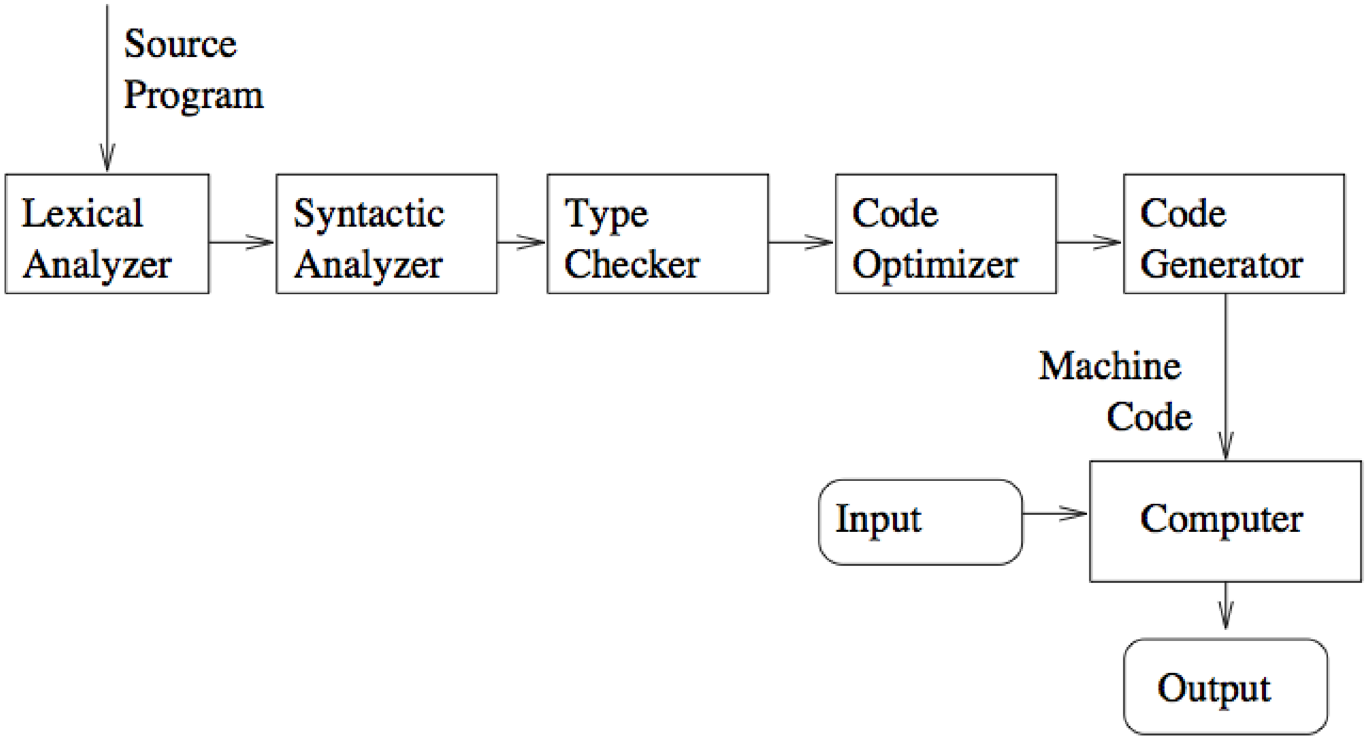


Figure 1.4: The Compile-and-Run Process

The Interpreting Process

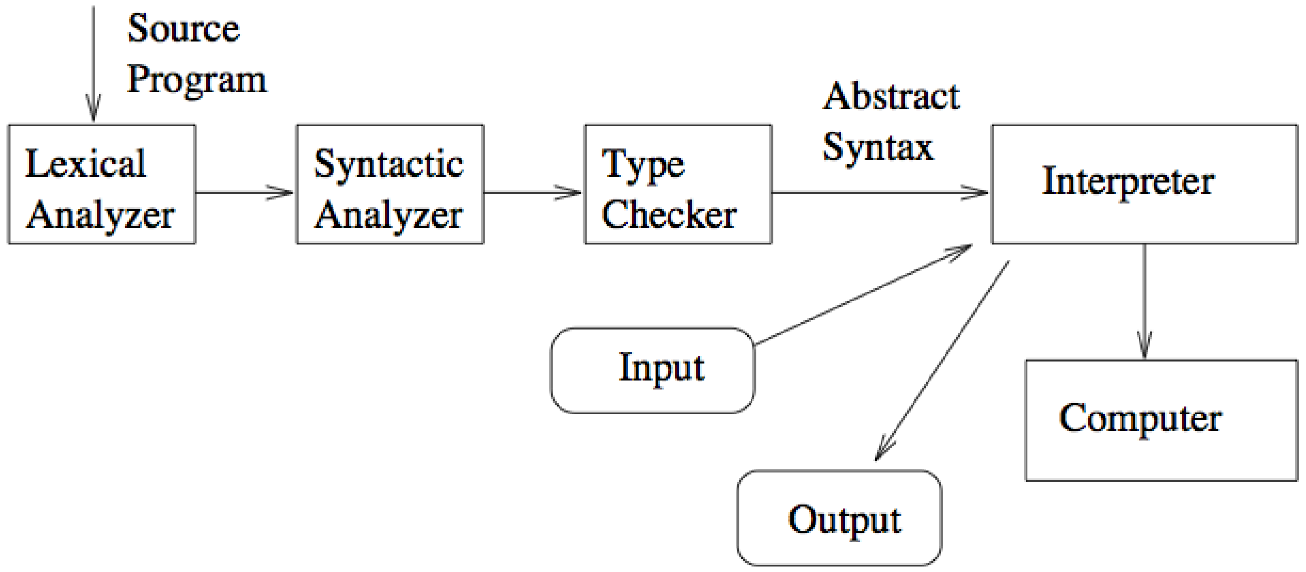


Figure 1.5: Virtual Machines and Interpreters

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: Logic Programming (Prolog)
- Case Study: Functional Programming (OCaml)

Finally, Enjoy This Video!

“The most popular programming languages 1965-2021”

https://www.bilibili.com/video/BV16t4y1B7Ji/