CSE216 Programming Abstractions

Programming Paradigms

YoungMin Kwon

Some UNIX commands

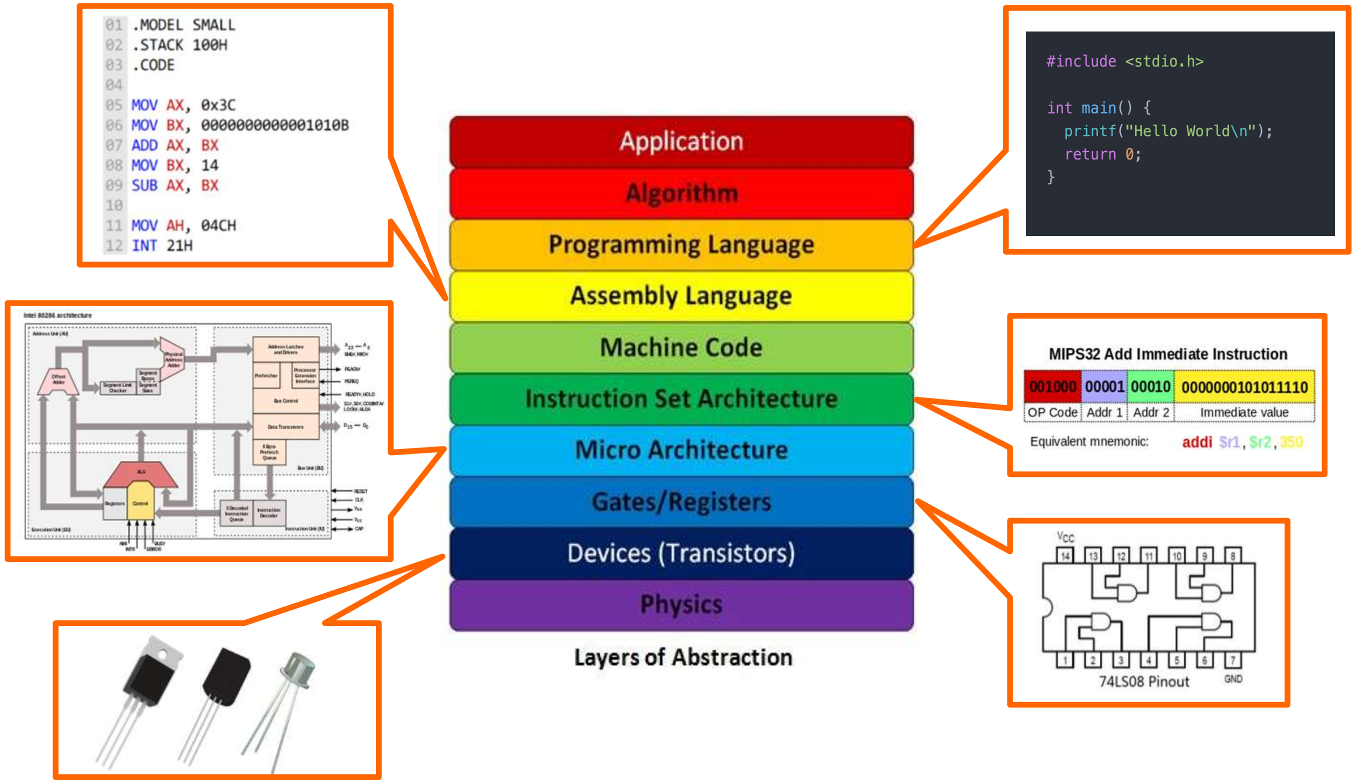
- About directories
  - ls: list directory contents.
    e.g. ls -al
  - pwd: print working directory.
    e.g. pwd
  - mkdir: make a directory.
    e.g. mkdir abc
  - cd: change directory.
    e.g. cd abc, cd ..
  - rmdir: remove a directory.
    e.g. rmdir abc

Some UNIX commands

- About files
  - cp: copy files.
    e.g. cp * abc/, cp a.txt b.txt
  - mv: move files.
    e.g. mv abc/* bcd/*, mv a.txt b.txt
  - cat: print the contents of a file.
    e.g. cat a.txt
  - grep: looking for a pattern.
    e.g. grep hello *
- man (manual page)
  - section number 2 is for system calls, 3 is for library routines
  - man 3 printf
  - man 2 fork
  - man sin

Abstractions

- Programming languages provide means of abstractions
  - Abstraction: hiding unwanted details and providing the most essential details



Layers of Abstraction

```
.MODEL SMALL
.STACK 100H
.CODE

MOV AX, 0x3C
MOV BX, 0000000000001010B
ADD AX, BX
MOV BX, 14
SUB AX, BX

MOV AH, 04CH
INT 21H
```

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

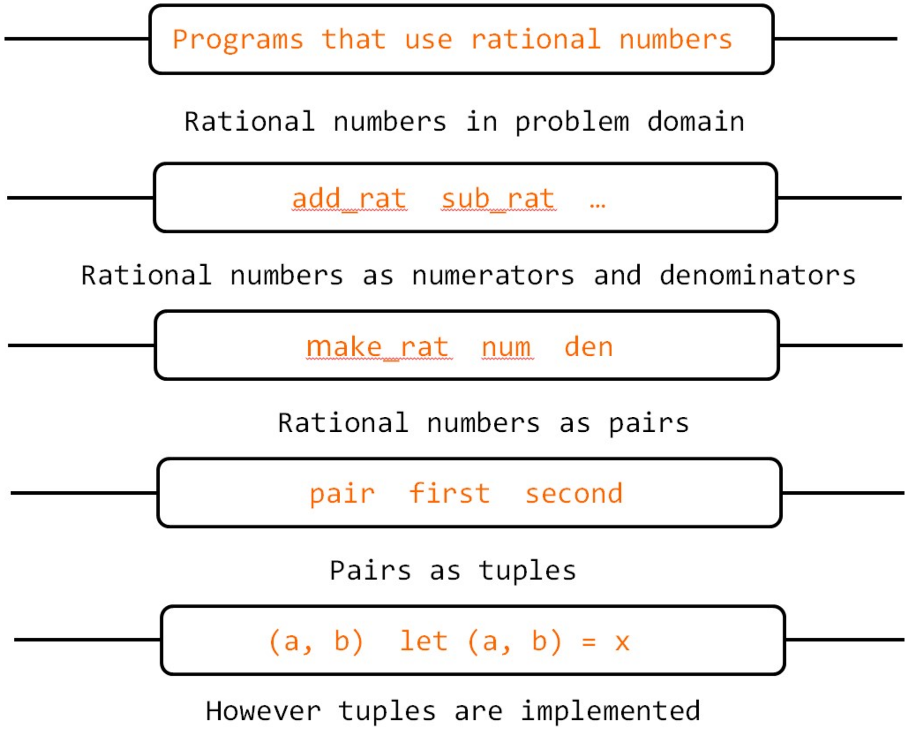
MIPS32 Add Immediate Instruction

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

Equivalent mnemonic: addi $r1, $r2, 350

Abstractions

- Abstractions in your program
  - To build a large program: build layers of abstractions



Programs that use rational numbers

Rational numbers in problem domain

add_rat sub_rat …

Rational numbers as numerators and denominators

make_rat num den

Rational numbers as pairs

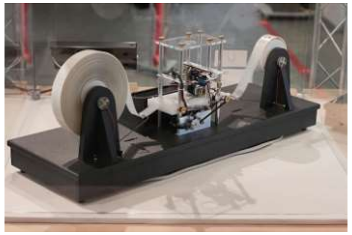
pair first second

Pairs as tuples

(a, b) let (a, b) = x

However tuples are implemented

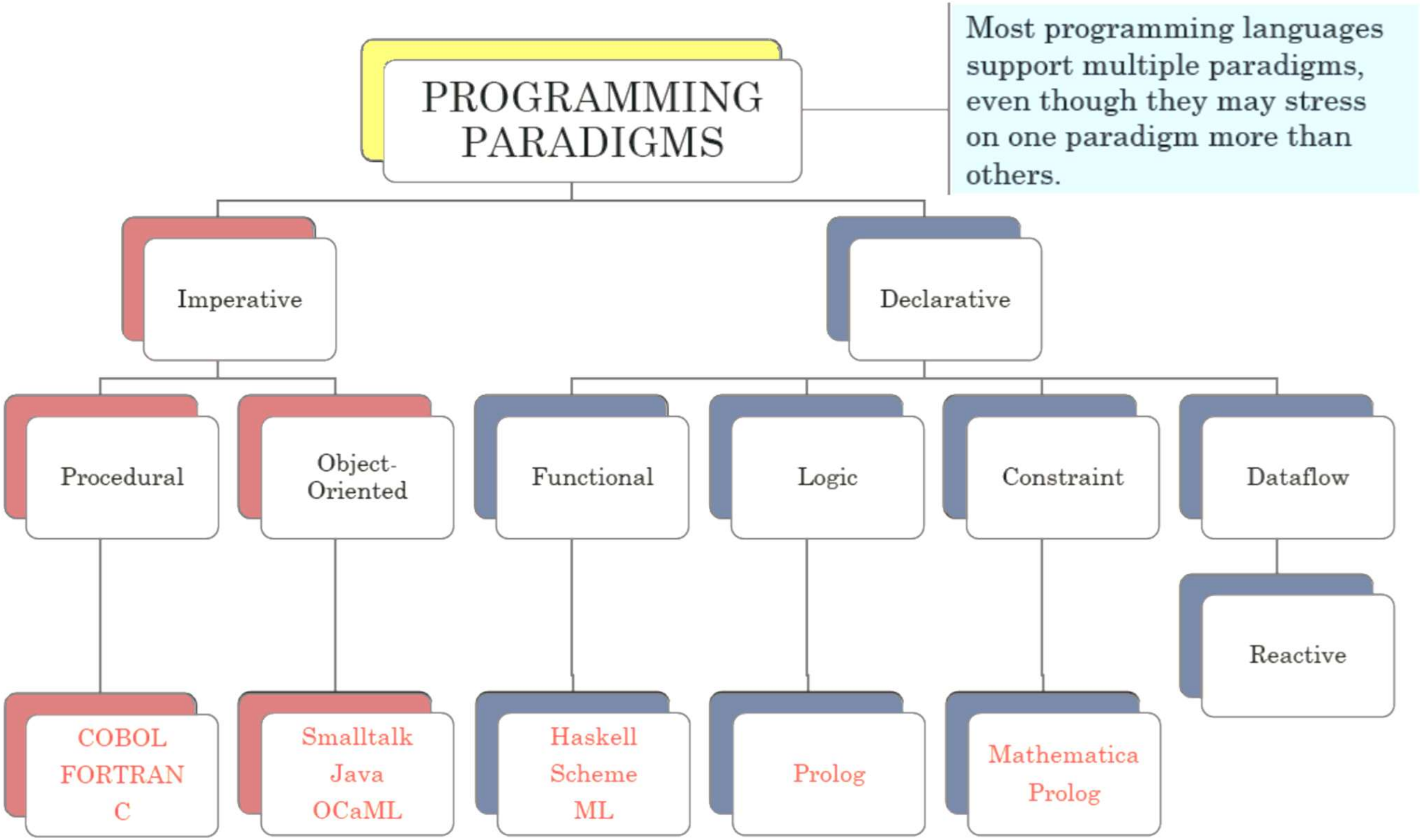
Programming Language Paradigms



- Imperative programming
  - Focus on how to achieve the goal
  - Update the state and take actions based on the state

- Declarative programming
  - Focus on describing what is the goal
  - Describe the logic of the program without specifying the order of evaluations

Programming Language Paradigms



Most programming languages support multiple paradigms, even though they may stress on one paradigm more than others.

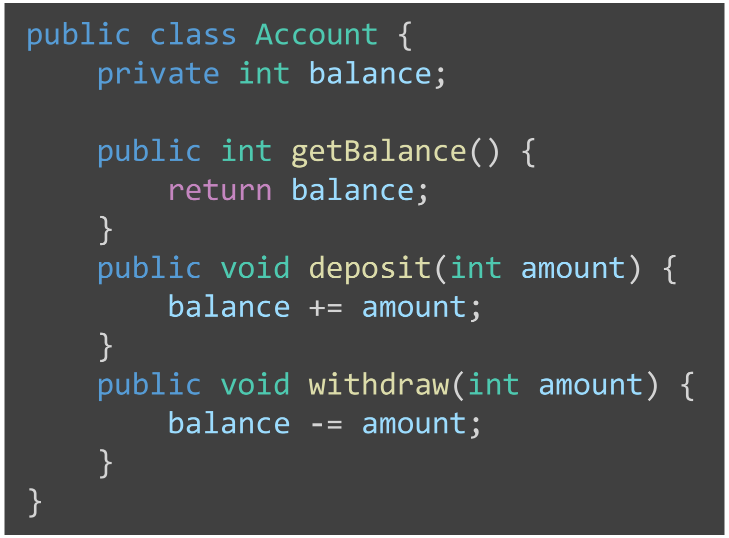
Procedural Programming

- Procedural programming
  - A kind of imperative programming
  - Abstraction mechanisms are procedures
  - COBOL, Fortran, C, Pascal
- Procedures
  - Contains a series of computational steps
  - State: local or global variables

Object Oriented Programming

- Object-oriented programming
  - A kind of imperative programming
  - A program comprises objects that interact with each other
  - C++, Java, OCaml, Smalltalk

- Objects
  - State: fields
  - Code: methods

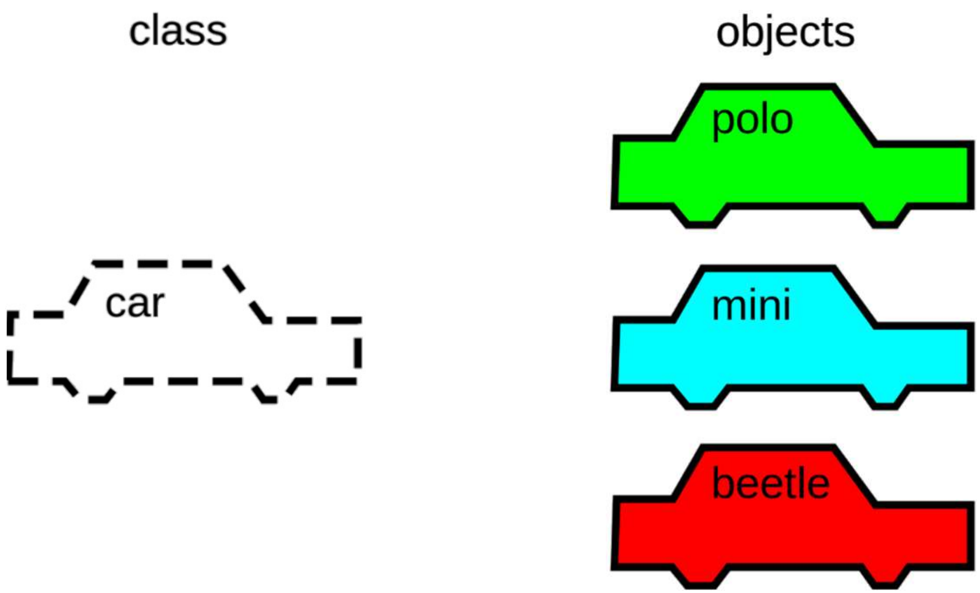


```java
public class Account {
    private int balance;

    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        balance += amount;
    }
    public void withdraw(int amount) {
        balance -= amount;
    }
}
```

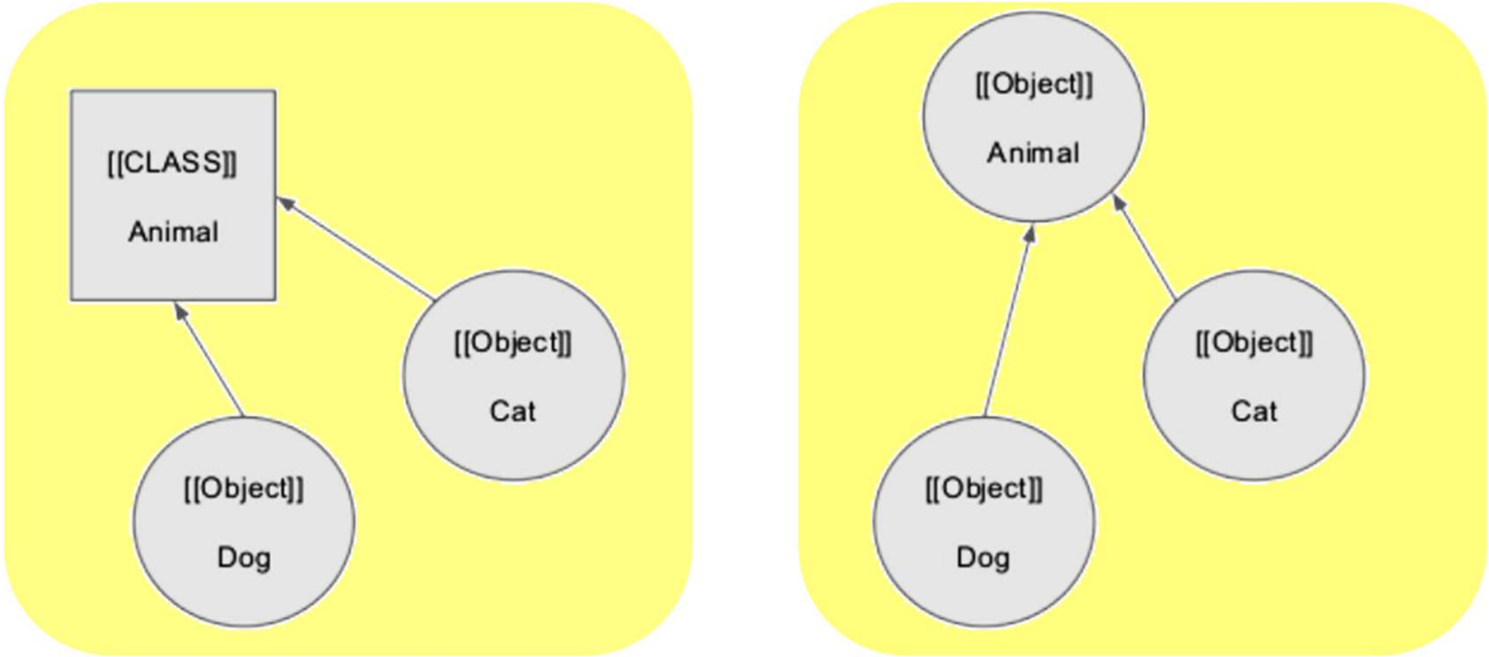
Object Oriented Programming

- Class-based
  - Class: definitions for the data format and procedures
  - Object: instance of a class



Object Oriented Programming

- Prototype-based
  - Objects have their own properties and methods
  - Objects delegate to their prototypes

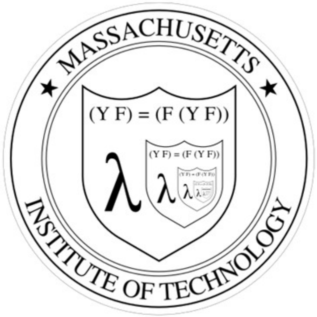


Object Oriented Programming

- Dispatching
  - Objects do select the method to run (not the external code)
  - Dynamic dispatching: decide the method to invoke at run time based on the object's actual type
- Message passing
  - Messages are exchanged between objects to communicate

Functional Programming

- Functional programming
  - Based on recursive definition of functions
  - Inspired from the lambda calculus developed by Alonzo Church



  - A program is viewed as a mathematical function that transforms an input to an output
  - Lisp, Scheme, ML, Haskell, …

Logic Programming

- Logic programming
  - Find solutions through logical rules and axioms
  - Goal: find a specific relation that is true by applying logical rules to axioms
  - Prolog
    - Prolog program: collection of rules (theorems) and facts (axioms)
    - Running a program: checks if a given query (goal) is provable from the axioms using the theorems

Prolog Example

```
/*simpsons.pl
*/

/*facts (axioms)*/
male(homer).
male(bart).

parent(homer, bart).
parent(homer, lisa).
parent(homer, maggie).
parent(marge, bart).
parent(marge, lisa).
parent(marge, maggie).

/*rules (theorems)*/
female(X)       :- \+ male(X). /*\+: not*/
child(C, P)     :- parent(P, C).
father(F, C)    :- parent(F, C), male(F).
mother(M, C)    :- parent(M, C), female(M).
son(S, P)       :- child(S, P),  male(S).
daughter(D, P) :- child(D, P),  female(D).
```

The Simpsons family

The Simpson family. From left to right: Bart, Santa's Little Helper, Marge, Maggie, Homer, Lisa, and Snowball II.

```
?- consult('simpsons.pl').
true.

?- father(homer, bart).
true .

?- mother(marge, bart).
true .

?- daughter(bart, marge).
false.

?- son(bart, marge).
true .

?- daughter(X, homer).
X = lisa ;
X = maggie.

?- halt.
```

GCD in Different Paradigms

- Imperative programming

```
int gcd(int a, int b) {
    while( a != b ) {
        if( a > b )
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

GCD in Different Paradigms

- Functional programming

```
let rec gcd a b =
    if a = b then a
    else if a > b then
        gcd (a - b) b
    else
        gcd (b - a) a
```

GCD in Different Paradigms

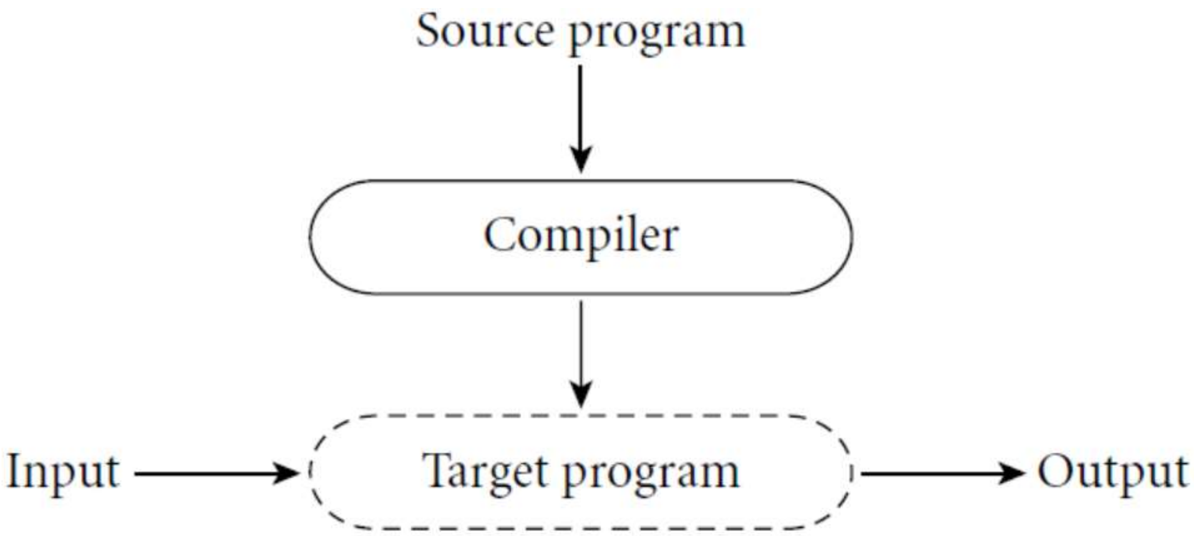
- Logic programming

```
gcd(A, B, G) :- A = B, G = A.
gcd(A, B, G) :- A > B, C is A - B, gcd(C, B, G).
gcd(A, B, G) :- B > A, C is B - A, gcd(C, A, G).
```

  - The proposition gcd(A, B, G) is true if
    - A, B, and G are all equal or
    - A > B and there is a number C such that C is A - B and gcd(C, B, G) is true or
    - B > A and there is a number C such that C is B - A and gcd(C, A, G) is true

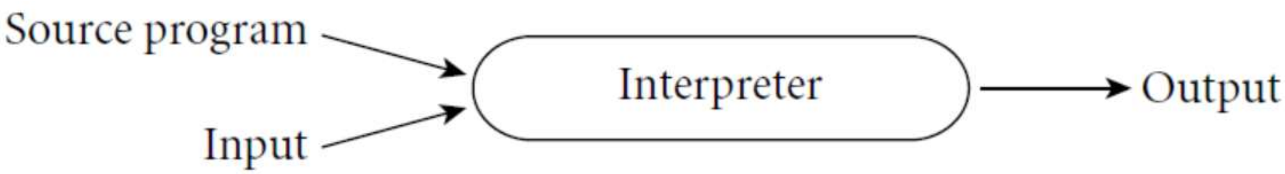
Compilation and Interpretation

- Pure compilation
  - Compiler translates high-level source programs into an equivalent target program
  - Later, the user tells the OS to run the program



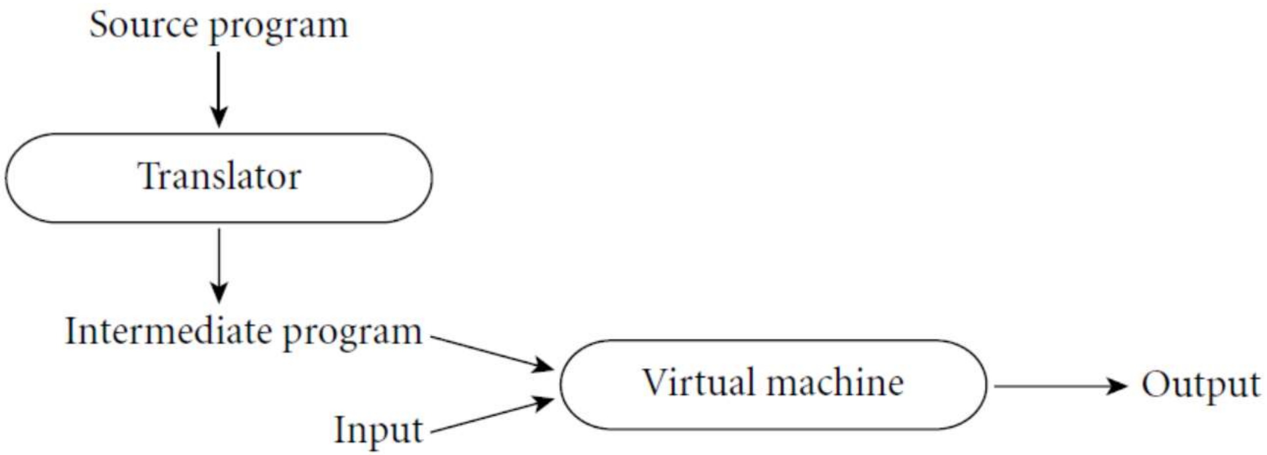
Compilation and Interpretation

- Pure interpretation
  - Interpreter implements a virtual machine
    - Its machine language is the high-level language
  - The interpreter reads the statements in that language and executes them



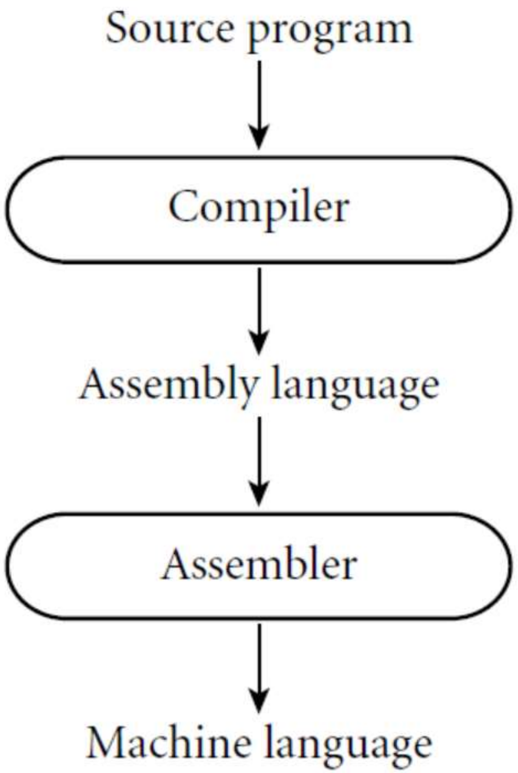
Compilation and Interpretation

- Mixing compilation and interpretation
  - A compiler generates an intermediate program
  - An interpreter reads the intermediate program and executes it



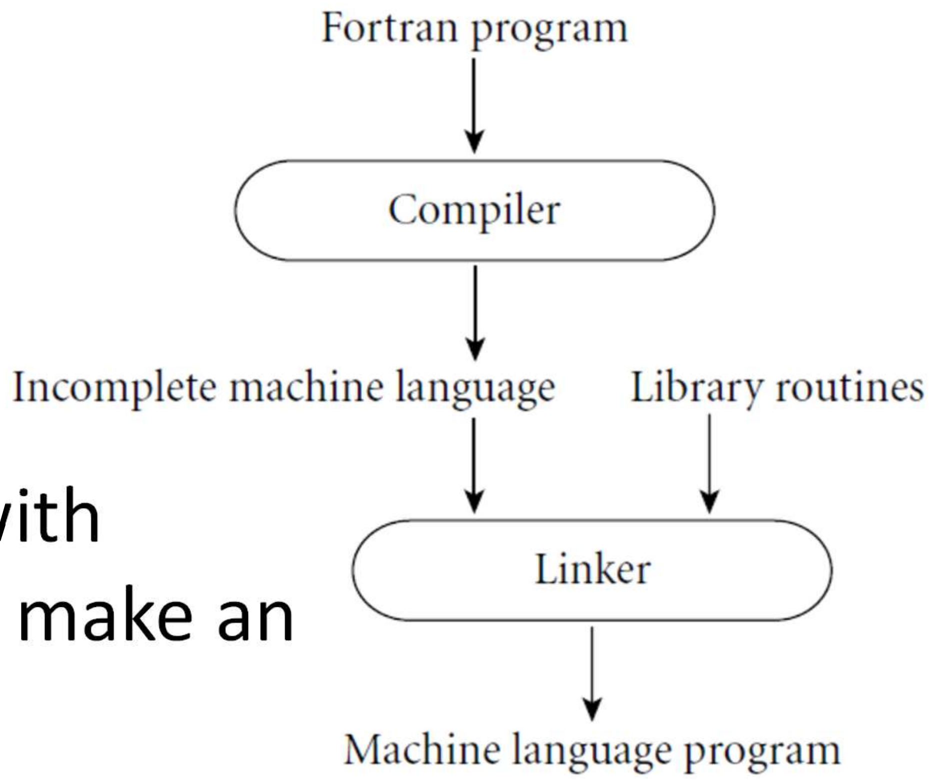
Compilation

- Many compilers generate assembly code
  - Assembler generates the machine code
  - Separates the source code from underlying h/w or OS changes



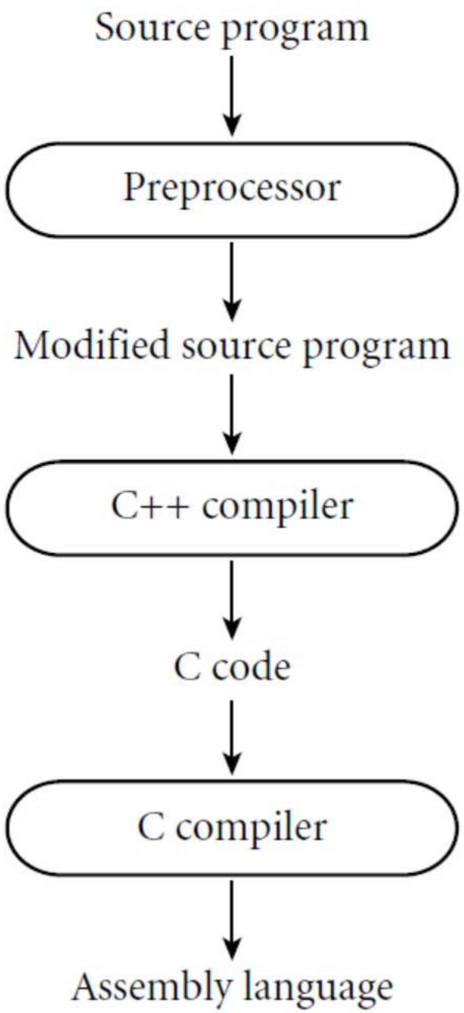
Compilation

- Linking library routines
  - Your program does not implement everything
    - E.g.) sin, cos, printf, …
  - Your program is linked with these library routines to make an executable object file



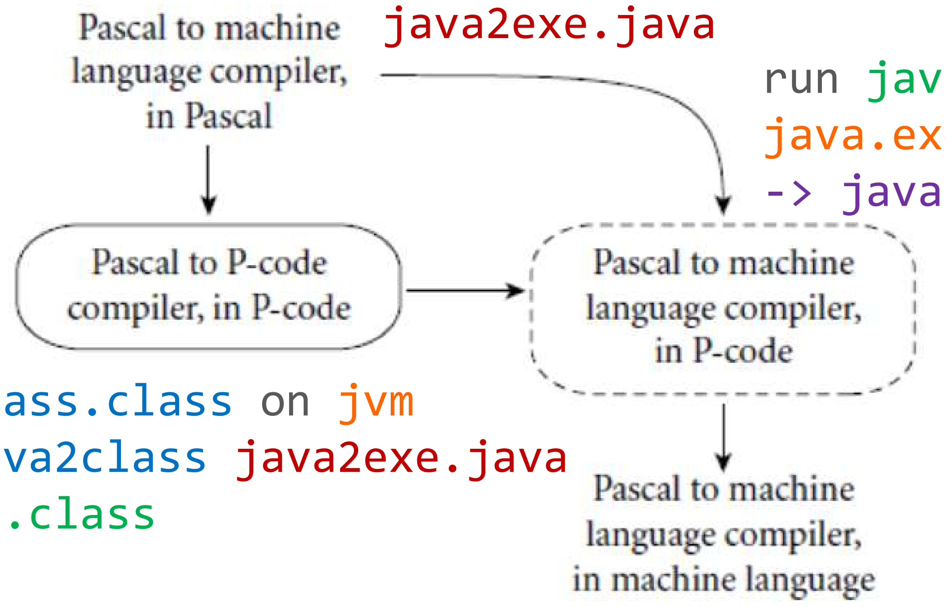
Compilation

- Source-to-source translation
  - AT&T's original C++ compiler
  - Generates C codes from C++ programs



Compilation

- How does one compile the first compiler?
  - Bootstrapping
    - Need only to implement P-code interpreter in machine language for each machine (e.g. jvm)
    - Need to implement Pascal to P-code compiler in P-code only once (e.g. java2class.class)



Pascal to machine language compiler, in Pascal
Pascal to P-code compiler, in P-code
Pascal to machine language compiler, in P-code
Pascal to machine language compiler, in machine language

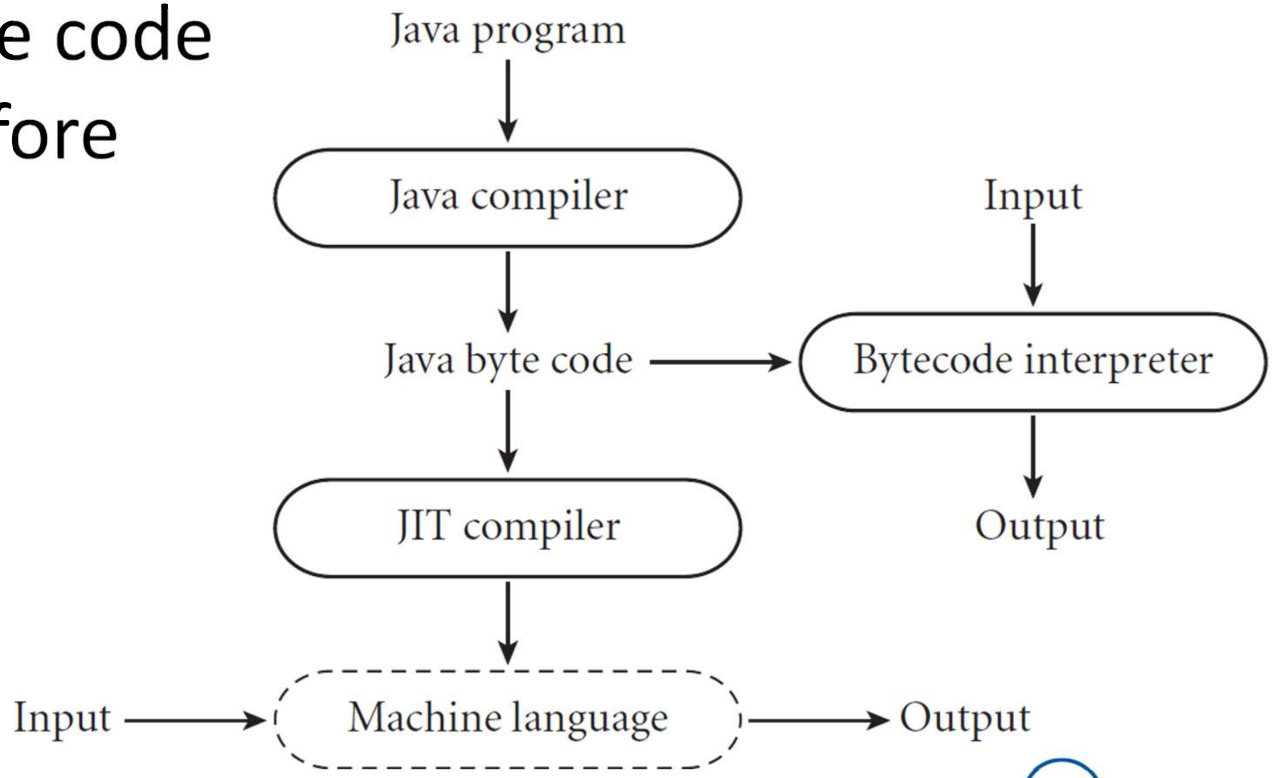
java2exe.java

```
run java2exe.class on jvm
java.exe java2exe java2exe.java
-> java2exe.exe
```

```
run java2class.class on jvm
java.exe java2class java2exe.java
-> java2exe.class
```

Compilation

- Just-In-Time (JIT) compilation
  - Java bytecode is a machine-independent code
  - The bytecode is translated into the machine code immediately before the execution



Java program
Java compiler
Input
Java byte code
Bytecode interpreter
JIT compiler
Output
Input
Machine language
Output