Introduction to Computer Science:

Programming Language Principle
– Python and C

May 2020

Honguk Woo

PL (programming language) course

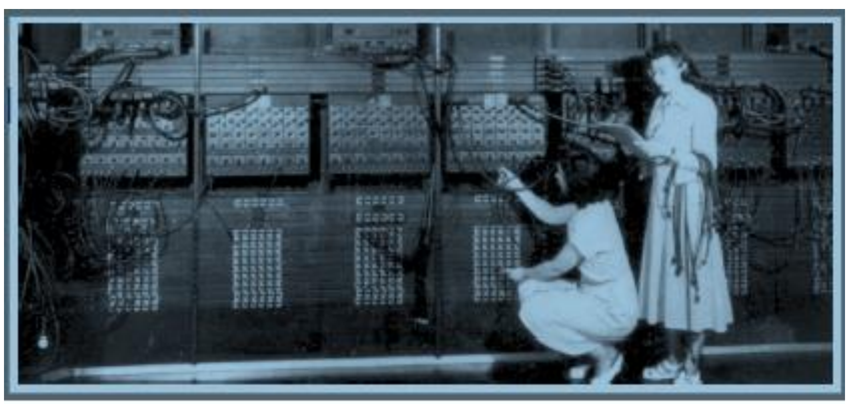
- Typically covers topics
  - Language concepts & Foundation
    - Language constructs : sequence, loop, conditions
    - Name, Scopes, Bindings
    - Subroutine and parameter passing
    - :
  - Language implementation
    - Common Tech used in compilers and interpreters
  - Various Language paradigms
    - Functional, Logic, OOP, Scripting

  :

- In this lecture, we will discuss
  - Some of those topics above briefly using Python and C and comparing them
    - *You might have some questions while you were working with Python and C this semester; why Python (or C) is designed as it is ?*
  - You will learn the topics in detail later in PL course

History of Programming language

- Early computing required rewiring for programming
  - *ENIAC (Electronic Numerical Integrator and Computer, 1946) programed with patch cords*
  - *Movie : imitation game*
- Von Neumann : stored-program computers
  - Innovation: **program is data**
  - Program stored in core memory
  - Allowed for “rapid” reprogramming.

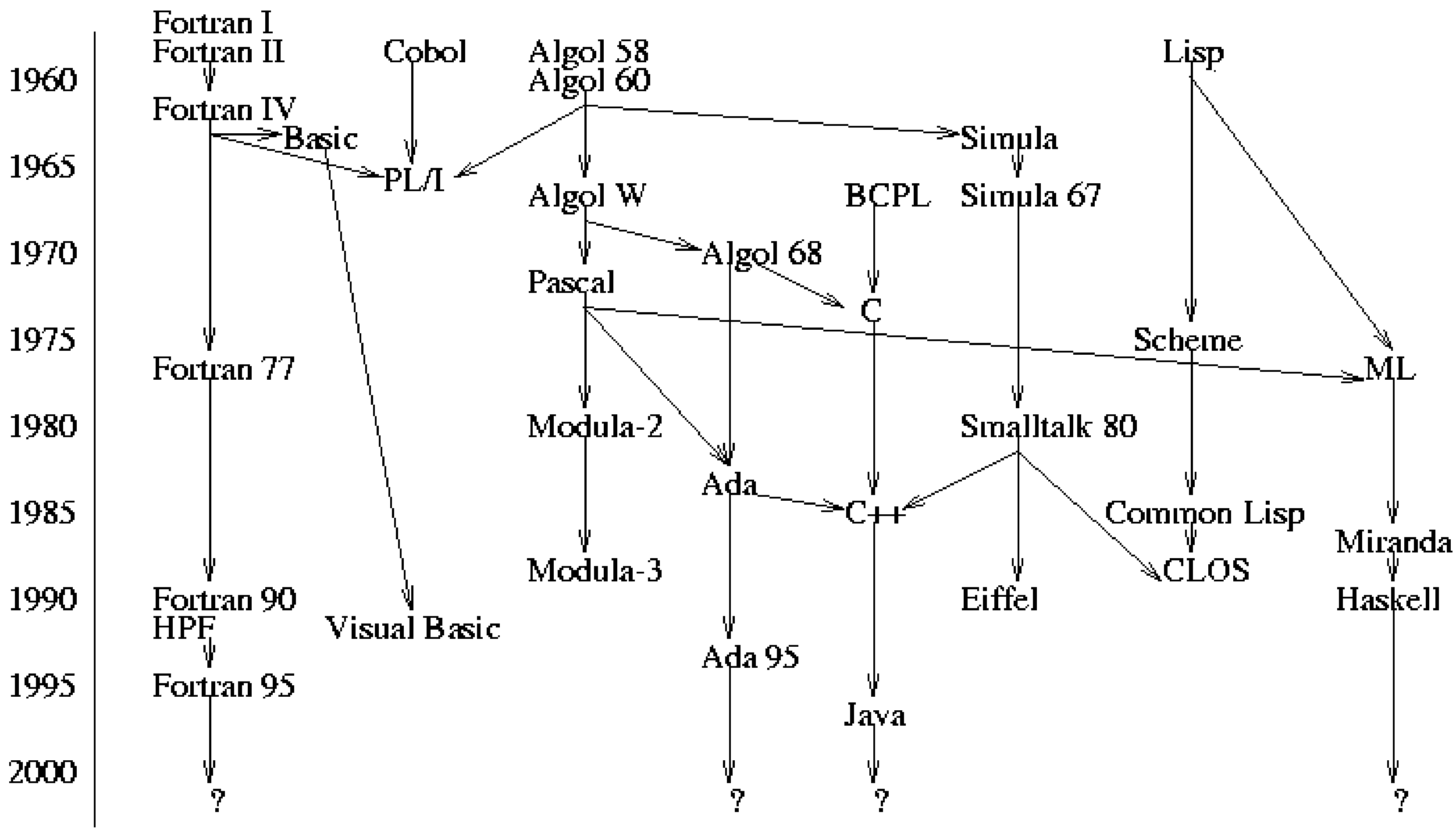


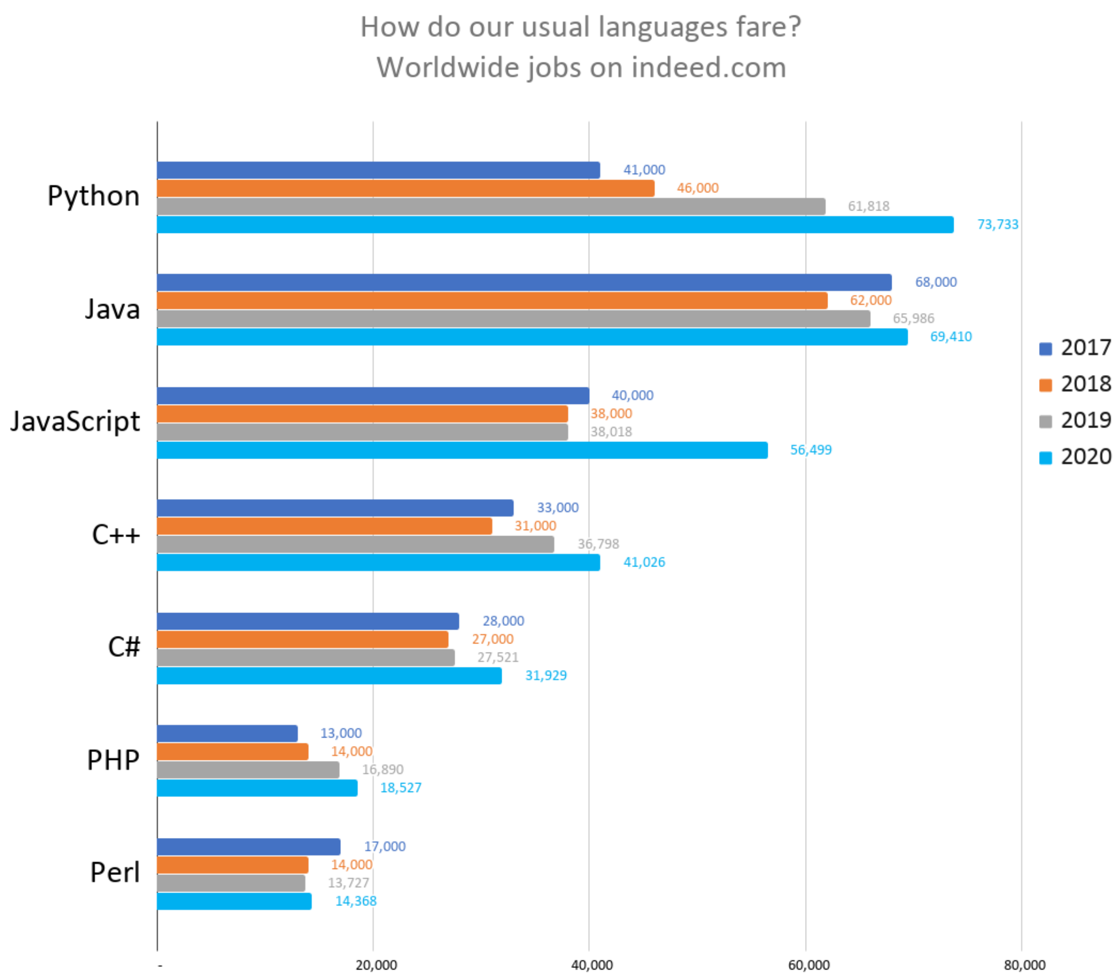
https://butlerscinemascene.com/2014/12/24/the-imitation-game/

Machine, Assembly, High level language

- Machine code
  - Sequence of binary numbers
  - Hard for humans to read and write, error-prone
- Assembly code
  - Computer transforms human-readable code into machine code
    - Assemble : mapping each line to its machine code equivalent
- High-level language
  - Assembly : still hard to read and write
    - Machine-specific, not portable
  - Desired higher-level, more human-friendly presentation
    - Machine-independent
    - Provides facilities for data and control flow abstraction
  - Translated to machine code via compiler (or interpreter)
    - Initially, slower than handwritten assembly code
    - Today, compiler generated code outperforms most human-written assembly code

```
   addiu   sp,sp,-32
   sw      ra,20(sp)
   jal     getint
   nop
   jal     getint
   sw      v0,28(sp)
   lw      a0,28(sp)
   move    v1,v0
   beq     a0,v0,D
   slt     at,v1,a0
A: beq     at,zero,B
   nop
   b       C
   subu    a0,a0,v1
:
```

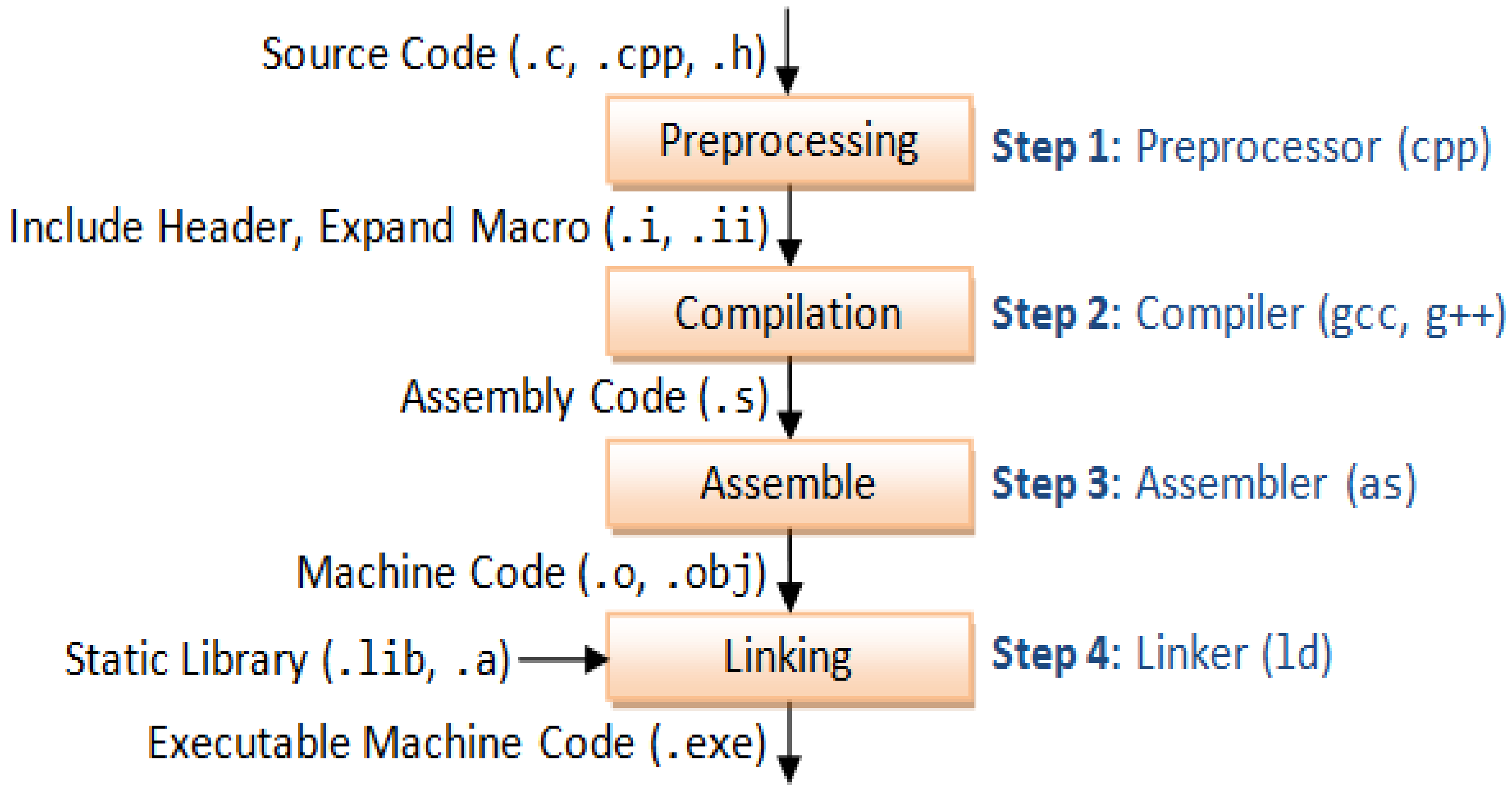

1960
1965
1970
1975
1980
1985
1990
1995
2000
Fortran I
Fortran II
Fortran IV
Basic
Cobol
PL/I
Algol 58
Algol 60
Algol W
Algol 68
Pascal
Simula
Simula 67
BCPL
C
Lisp
Scheme
ML
Fortran 77
Modula-2
Smalltalk 80
Ada
C++
Common Lisp
Miranda
Modula-3
CLOS
Eiffel
Haskell
Fortran 90
HPF
Visual Basic
Ada 95
Fortran 95
Java
?
?
?
?



https://www.codingdojo.com/blog/top-7-programming-languages-of-2020

- Python
- C/C++
- Java
- Javascript
- :
- :

C (gcc in Linux) compilation procedure



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

hworld.c

```
#include <stdio.h>

int main() {
  printf("hello world!\n");
  return 0;
}
~
~
~
```

compile

```
hong@Ubuntu-V: ~/myLec/2018.s/SP/0_hello
hong@Ubuntu-V:~/myLec/2018.s/SP/0_hello$ gcc -O0 --save-temps -o hworld hworld.c
hong@Ubuntu-V:~/myLec/2018.s/SP/0_hello$ ls -al
total 52
drwxrwxr-x 2 hong hong  4096  3월  5 14:32 .
drwxrwxr-x 3 hong hong  4096  3월  5 10:10 ..
-rwxrwxr-x 1 hong hong  8608  3월  5 14:32 hworld
-rw-rw-r-- 1 hong hong    75  3월  5 10:12 hworld.c
-rw-rw-r-- 1 hong hong 17109  3월  5 14:32 hworld.i
-rw-rw-r-- 1 hong hong  1504  3월  5 14:32 hworld.o
-rw-rw-r-- 1 hong hong   457  3월  5 14:32 hworld.s
hong@Ubuntu-V:~/myLec/2018.s/SP/0_hello$ ./hworld
hello world!
hong@Ubuntu-V:~/myLec/2018.s/SP/0_hello$
```

hworld.i

```
hong@Ubuntu-V: ~/myLec/2018.s/SP/0_hello
# 912 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 942 "/usr/include/stdio.h" 3 4

# 2 "hworld.c" 2


# 3 "hworld.c"
int main() {
  printf("hello world!\n");
  return 0;
}
~
```

hworld.s

```
        .file   "hworld.c"
        .section        .rodata
.LC0:
        .string "hello world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
```

hworld.o

```
hong@Ubuntu-V: ~/myLec/2018.s/SP/0_hello
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@ ^B^
@^@^@^@^@^@^@^@^@@^@^@^@^@^@@^@^M^@
^@UH<89>å¿^@^@^@^@è^@^@^@^@¸^@^@^@^@]Ãhello world!^@^@GCC: (Ubuntu 5.4.0-6ubuntu
1~16.04.4) 5.4.0 20160609^@^@^T^@^@^@^@^@^@^@^AzR^@^Ax^P^A^[^L^G^H<90>^A^@^@^\^@
^@^@^\^@^@^@^@^@^@^@^U^@^@^@^@A^N^P<86>^BC^M^FP^L^G^H^@^@^@^@^@^@^@^@^@^@^@^@^
@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@ñÿ^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^
@^C^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^C^@^@^@^@^@^@^@^@^@^@^@^@^
@^@^@^@^@^@^@^@^C^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^E^@^@^@^@^
@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^G^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^
@^C^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^F^@^@^@^@^@^@^@^@^@^@^@^
@^@^@^@^@
^@^@^@^R^@^A^@^@^@^@^@^@^@^@^@^U^@^@^@^@^@^@^O^@^@^@^P^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@hworld.c^@main^@puts^@^@^@^@^@^E^@^@^@^@^@^@^@
^@^@^@^E^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^B^@^@^@
@
@
@
@
@
@
@
@
@
"hworld.o" [Incomplete last line][converted] 6 lines, 1528 characters
```

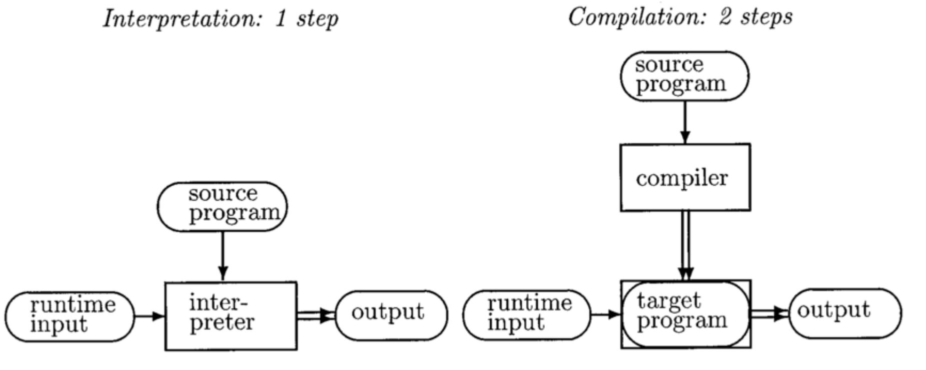
```
hong@Ubuntu-V:~/myLec/2018.s/SP/0_hello$ objdump -d hworld.o

hworld.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   bf 00 00 00 00          mov    $0x0,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   5d                      pop    %rbp
  14:   c3                      retq
```

Various language paradigms

- There are many programming languages:
  - Imperative languages : focus on **how** the computer should do
    - Procedural : **C**, Fortran, Pascal
      - Evolving from assembly; how the computer works internally
      - A program is a sequential computation that directly manipulates simple typed data (memory locations)
    - Object-oriented : C++., Java, **Python**
      - Human-inspired model; objects encapsulate all related state (data)
      - Complex problems can be decomposed
    - Scripting : **Python**, Shell, Perl
      - Focus on “developer productivity” (rapid development)
      - Could be procedural, OOP, functional, …
  - Declarative languages : focus on **what** the computer should do
    - Functional : Lisp, Haskell
      - Mathematics-inspired model; computer job is to compute the result of applying the program (function) to the input
      - A program is defined in terms of mathematical functions
    - Logic : Prolog

Compilation, Interpretation

- In general, a processor can only execute machine code; thus high level code should be translated for execution
  - Compilation : ahead of execution time
    - Translation occurs only once but program is executed many times
  - Interpretation : piece-wise during execution time
    - Each execution requires on-the-fly translation
  - C : requires compilation, e.g., *gcc -o helloworld helloword.c*
  - Python : requires Interpretation



  - Advantage / Disadvantage of Compilation ?
  - Advantage / Disadvantage of Interpretation ?

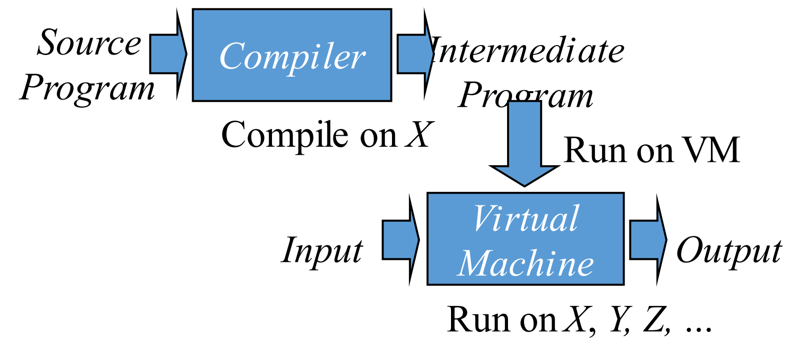
Compilation, Interpretation

- **Compilation :**
  - Resulting program executes faster than if interpreted
  - Requires code generation and detailed platform knowledge

- **Interpretation :**
  - Programming language can be much more flexible
  - Can be portable.
  - Inefficient

Mixed Compilation and Interpretation

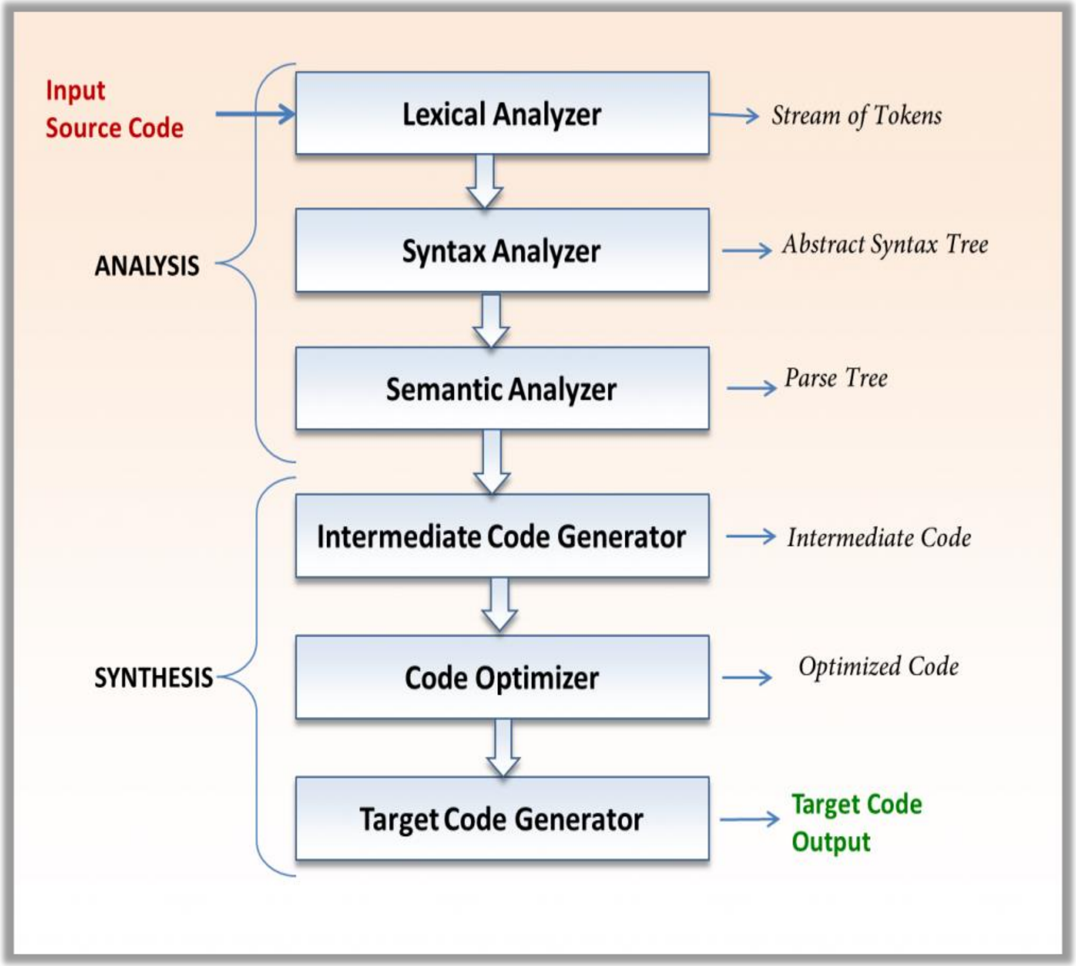
- **Interpreting high-level languages is usually slow**
  - First compile high-level to low-level byte code
  - Interpret much simpler byte code
    - Byte code is executed by **virtual machine**



- Explicit compilation
  - Separate compilation step
  - User is aware of byte code (e.g., Java)
    - Java : javac (compiler), JVM (virtual machine)
- Implicit compilation
  - Tool appears as interpreter to user
  - Compilation occurs “behind the scenes”
  - Compilation only required once if byte code is cached (e.g., Python)

Compilation Process

① Lexical analysis
② Syntax analysis
③ Semantic analysis
④ Intermediate code generation
⑤ Target code generation
⑥ Optimization



https://www.kttpro.com/2017/02/09/six-phases-of-the-compilation-process/

Name, Binding

- Name
  - An identifier, made up of a string of characters
  - Allow us to refer entities in a program by a symbol instead of an address
  - Provide a level of abstraction in a program
    - Class (object) for data abstraction
    - Function for control abstraction

- Binding
  - Associate a name with some entity (e.g., object)
  - Binding a name to an entity resolves an abstraction
  - Binding time
    - When is a name resolved ? (e.g., compilation time, run time)
    - Static vs Dynamic (late) binding
  - Polymorphism
    - A name can be bound to more than one entity

Name, Binding

- Python : bind a name to an object using assignment

  a = 1

  b = 2

  c = a

  - Bind the name c to the object bound to the name a
    - A name can only be bound to an object, not to a name

```
a = [1,2]
b = a
a.append(3)
b = [4]
```

what's a and b ?

A name bound to a mutable object in Python is a pointer in C ?

Name, Binding

Python : **dynamically typed names**

- Only values (objects) have a fixed type

```
x = 1           # x is an integer
x = 'hello'     # now x is a string
x = [1, 2, 3]   # now x is a list
```

In Python, variables are a name that refers to a value (object) in memory

C : **statically typed variables with declaration**

- Each variable has a fixed type
- Type checking at compilation time

```
int x = 4;
char y = 'a';
char* p = &y;
```

In C, variables are a location in the memory where values are stored; in fact, C has pointer types for such Python variables.

Discussion : Python and C

```
x = 10
y = x
x += 5
print("x =", x)
print("y =", y)
```

```
x = [10, 0]
y = x
x[1] = 5
print("x =", x)
print("y =", y)
```

```
int x = 10;
int y = x;
x += 5
printf("x =%d\n", x)
printf("y =%d\n", y)
```

```
Int x[] = {10, 0};
Int* y = x;
x[1] = 5;
printf(“x[1]=%d\n”, x[1]);
printf(“y[1]=%d\n”, y[1]);
```

Binding time

- When is meaning of “+” bound to its meaning in “x + 10”?
  - Could be at language definition, implementation, or at translation
  - May also be execution time — could depend on type of x determined at run-time
- OOP
  - Binding time of methods ?

```
a = Animal("mydog")
b = Dog("mydog2")
c = Duck("myduck")
a.move()
?
b.move()
?
c.move()
?
b.speak()
?
c.speak()
?
a.speak()
?
```

```
class Animal:
    def __init__(self, name):
        self.name = name
    def move(self):
        print("move")
    def speak(self):
        pass

class Dog (Animal):
    def speak(self):
        print("bark")

class Duck (Animal):
    def speak(self):
        print("quack")
```

C Pointer

- Python (and modern high level languages) hides how the memory in computer is actually organized
  - *Did you try id() of python objects ?*
- *C* provides more computer-friendly view on memory; C expects you to have a basic understanding that memory consists of a sequence of bytes

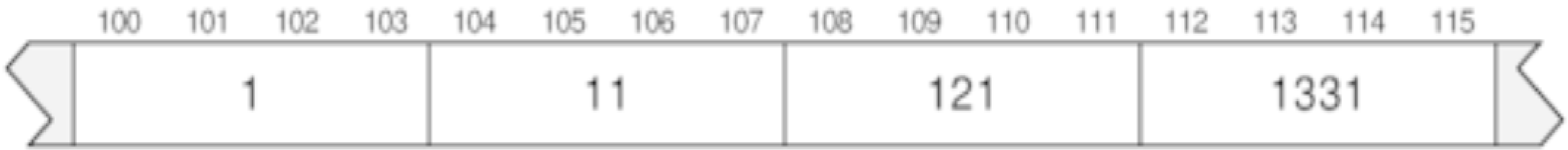
```
struct foo { char c; int i; };
struct foo my_array[3] = {
    { 'x', 123 },
    { 'y', 456 },
    { 'z', 789 },
};
```

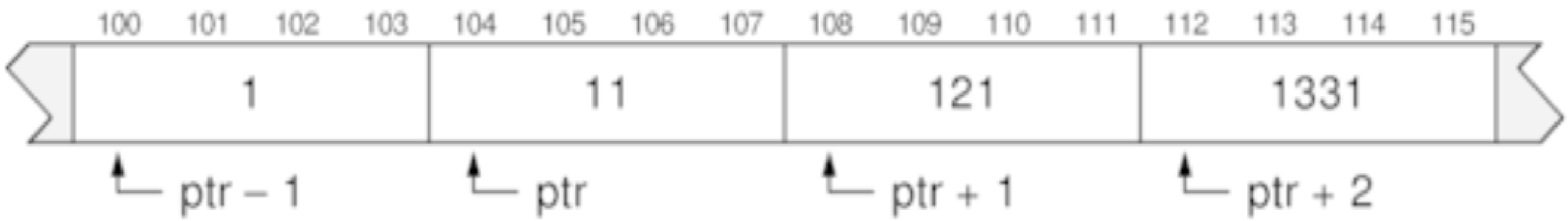
So, pointer arithmetic works as
p = address of my_array
p = p + 1

C Pointer

- Pointer arithmetic in C
  - Note that p + 1 for pointer is not same as p + 1 for integer
  - For pointer, address incremented by size of object pointed to (e.g., array indexing)

```
int array[4] = { 1, 11, 121, 1331 };
int *ptr = &array[1];  /* ptr points to the second element (11) */
```





Q : where the variables in C are stored ?

- Stack
- Data Segment
- Heap

```
#include <stdio.h>
#include <stdlib.h>
int x;

int main(void) {
  int y;
  char *str;
  y = 4;

  :
  str = malloc(100*sizeof(char));
  :

  free(str);
  return 0;
}
```

- *Who did "malloc" and "free" for you in python ?*

Parameter Passing

- In C
  - **Call by value** : values in actual parameters (caller) are copied into the formal parameters (callee); different memory is allocated for actual and formal parameters
  - **Call by reference** : the address of the variable is passed to the function (the address is copied to the formal parameters, as same as the above)
- In Python
  - Based on **name binding (**similar to call by reference, but**)**
    - Immutable object : cannot be modified
    - Mutable object : ?

```
def changeme( mylist ):
  mylist.append([1,2,3,4])
  print (mylist)
  return

mylist = [10,20,30];
changeme( mylist );
print (mylist)
```

```
def changeme( mylist ):
  mylist = [1,2,3,4]
  print (mylist)
  return

mylist = [10,20,30];
changeme( mylist );
print (mylist)
```

Parameter Passing

Call by Value

```
void swapByValue(int a, int b){
  int t;
  t = a; a = b; b = t;
}


int main(){
  int n1 = 10, n2 = 20;
  swapByValue(n1, n2);
  printf("n1: %d, n2: %d\n", n1, n2);
}
```

Call by Reference

```
void swapByReference(int *a, int *b){
  int t;
  t = *a; *a = *b; *b = t;
}


int main() {
  int n1 = 10, n2 = 20;
  swapByReference(&n1, &n2);
  printf("n1: %d, n2: %d\n", n1, n2);
}
```

Parameter Passing

Call by Value or Reference ?

```
#include <stdio.h>
void display(int age) {
  printf("%d", age);
}

int main() {
  int ageArray[] = {2, 3, 4};
  display(ageArray[2]);
  return 0;
}
```

```
#include <stdio.h>
float average(float age[]);

int main() {
  float avg, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
  avg = average(age);
  printf("Average age = %.2f", avg);
  return 0;
}

float average(float age[]) {
  int i;
  float avg, sum = 0.0;
  for (i = 0; i < 6; ++i) {
     sum += age[i];
   }
  avg = (sum / 6);
  return avg;
}
```

- Python
  - Assignment is “Binding” : bind name to object, e.g., a = [10, 11, 12]
  - Mutation is not “Binding” : e.g., a.append(3)

- C
  - Pointer is “Binding” like python (reference), e.g, int* p = &i;
  - Non-pointer is “Binding” like container, e.g., int i = 3;

```
total += 1
```

In python,
is it mutation or not ?

What about C ?

Q : output ?

```
def f1(p):
    p.extend([4])

def f2(p):
    p += [4]

def f3(p):
    p = p + [4]


a = [1, 2, 3]
f1(a)
print(a)

a = [1, 2, 3]
f2(a)
print(a)

a = [1, 2, 3]
f3(a)
print(a)
```

```
def f1(p):
    p = p[:]

def f2(p, q):
    p = p[:]
    q = p

def f3(p):
    p = p[:]+[4]
    return p

a = [1, 2, 3]
f1(a)
print(a)

a = [1, 2, 3]
q = 4
f2(a, q)
print(q)

a = [1, 2, 3]
a = f3(a)
print(a)
```

Discussion

- *Why does C support array of same data type elements ?*
- *Why does Python list support array of different data type elements ?*