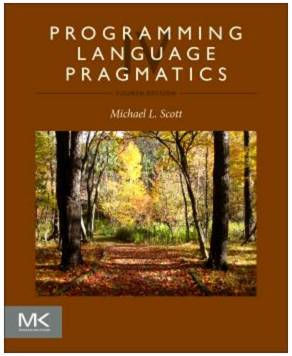
Chapter 1 :: Introduction

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Definitions

- A **program** is an expression of an algorithm, encoded for execution on a machine.

- A **programming language** is an artificial language with its own rules of syntax, used for expressing programs.

- A **programmer** is a person who uses programming languages to design programs and works to get them to run without error on machines.

Evolution

- The **programmers** who used the first electronic computers believed that the computer’s time was more valuable than theirs.
- They programmed in machine language.

Evolution

- **Machine language**

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

GCD program in x86 machine language

Evolution

- **Machine language**
- **Assembly language** in the form of one-to-one correspondences between mnemonics and machine language instructions.

```
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        andl    $-16, %esp
        call    getint
        movl    %eax, %ebx
        call    getint
        cmpl    %eax, %ebx
        je      C
A:      cmpl    %eax, %ebx
        jle     D
        subl    %eax, %ebx
B:      cmpl    %eax, %ebx
        jne     A
C:      movl    %ebx, (%esp)
        call    putint
        movl    -4(%ebp), %ebx
        leave
        ret
D:      subl    %ebx, %eax
        jmp     B
```

GCD program in x86 assembly language

Evolution

- **Machine language**
- **Assembly language** in the form of one-to-one correspondences between mnemonics and machine language instructions.
- Mnemonics to mathematical formulae (Fortran).

Evolution

- **Machine language**
- **Assembly language** in the form of one-to-one correspondences between mnemonics and machine language instructions.
- Mnemonics to mathematical formulae (Fortran).
- From assembly language to **machine-independent languages**.

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

GCD program in C

Why Are There So Many Programming Languages?

- Evolution -- we've learned better ways of doing things over time
  - goto based control flows → loops → nested block structures → object-orientation

- Special purposes -- some languages were designed for a specific problem domain
  - C: low-level systems programming
  - Prolog: reasoning about logical relationships

- Personal preference -- diverse ideas about what is pleasant to use

What Makes a Language Successful?

- Expressive power
- Ease of use for the novice -- Basic, Logo
- Ease of implementation
- Standardization
- Open source
- Excellent compilers
- Economics, patronage (Objective-C as the official language for iPhone and iPad apps)

The Programming Language Spectrum

- Declarative
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, Constraint-based (Prolog, VisiCalc, RPG)
- Imperative
  - Von Neumann (Fortran, Pascal, Basic, C)
  - Object-oriented (Smalltalk, Eiffel, C++, Java)
  - Scripting languages (Perl, Python, JavaScript, PHP)

The GCD Algorithm

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

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

gcd(A,B,G) :- A = B, G = A.                      % Prolog
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).
```

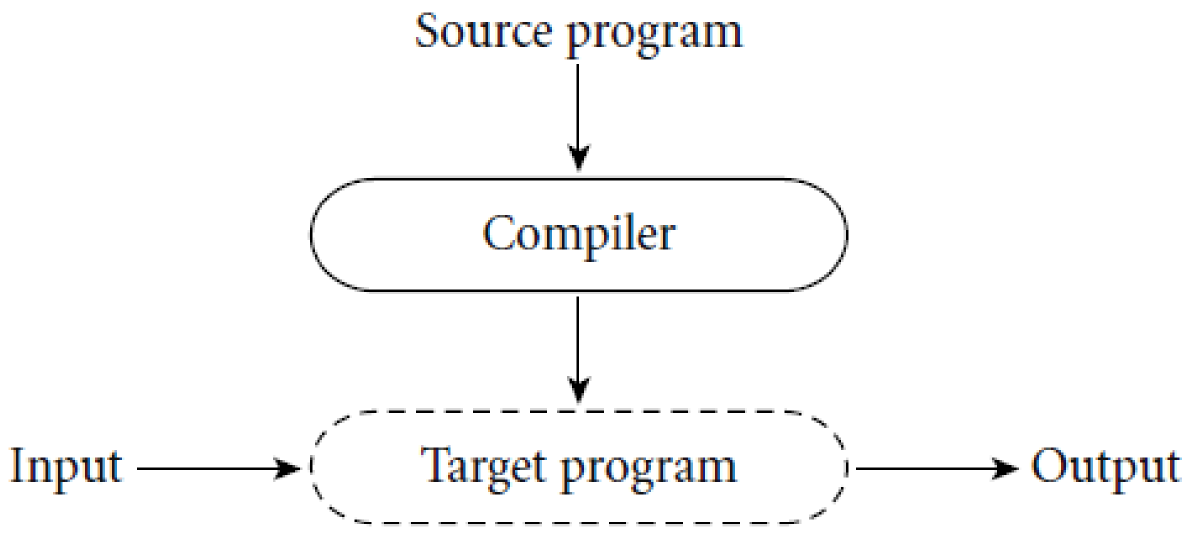
Figure 1.2 The GCD algorithm in C (top), OCaml (middle), and Prolog (bottom). All three versions assume (without checking) that their inputs are positive integers.

Why Study Programming Languages?

- Help you choose a language.
- Make it easier to learn new languages.
- Understand obscure features.
- Make good use of debuggers.
- Simulate useful features in languages that lack them.
- Make better use of language technology wherever it appears.

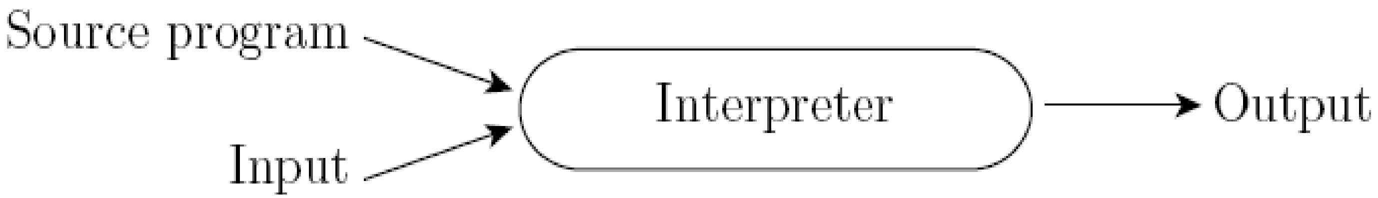
Compilation

- Compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away.
- At some arbitrary later time, the user tells the operating system to run the target program.



Interpretation

- Unlike a compiler, an interpreter stays around for the execution of the application.
- Interpreter is the locus of control during execution.



Compilation vs. Interpretation

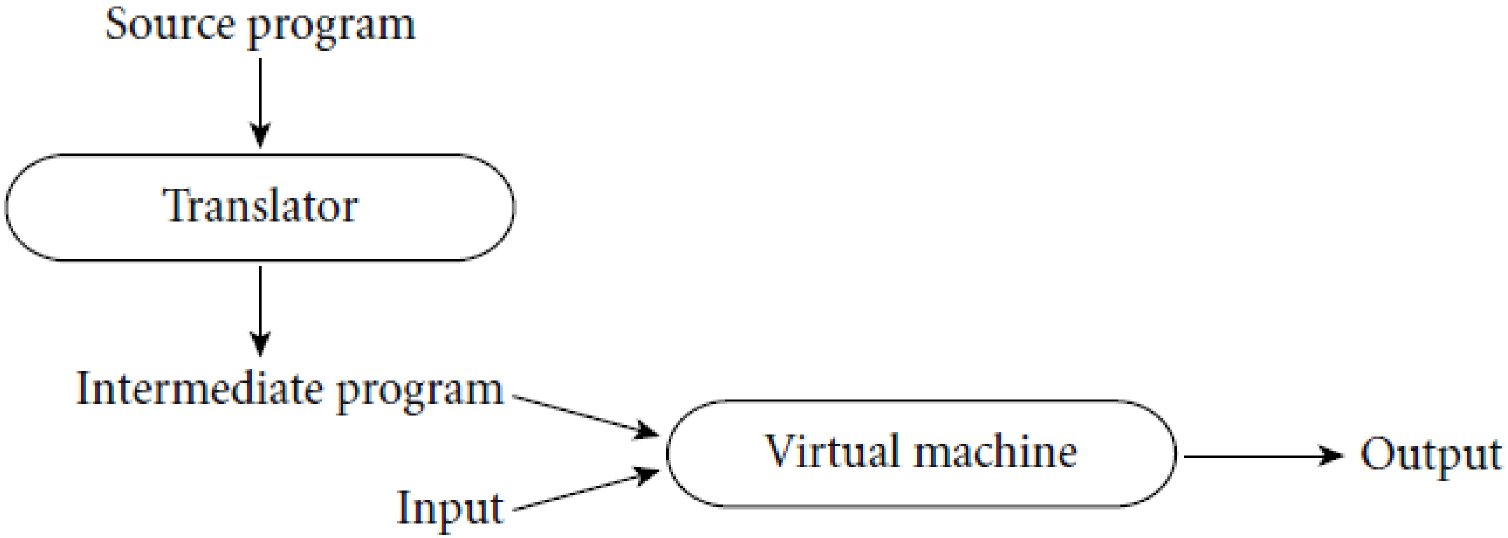
- Compilation vs. interpretation
  - Not opposites
  - Not a clear-cut distinction

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation:
  - Better performance

Mixing Compilation and Interpretation

- Most language implementations include a mixture of both compilation and interpretation
- Common case is compilation or simple pre-processing, followed by interpretation



Phases of Compilation