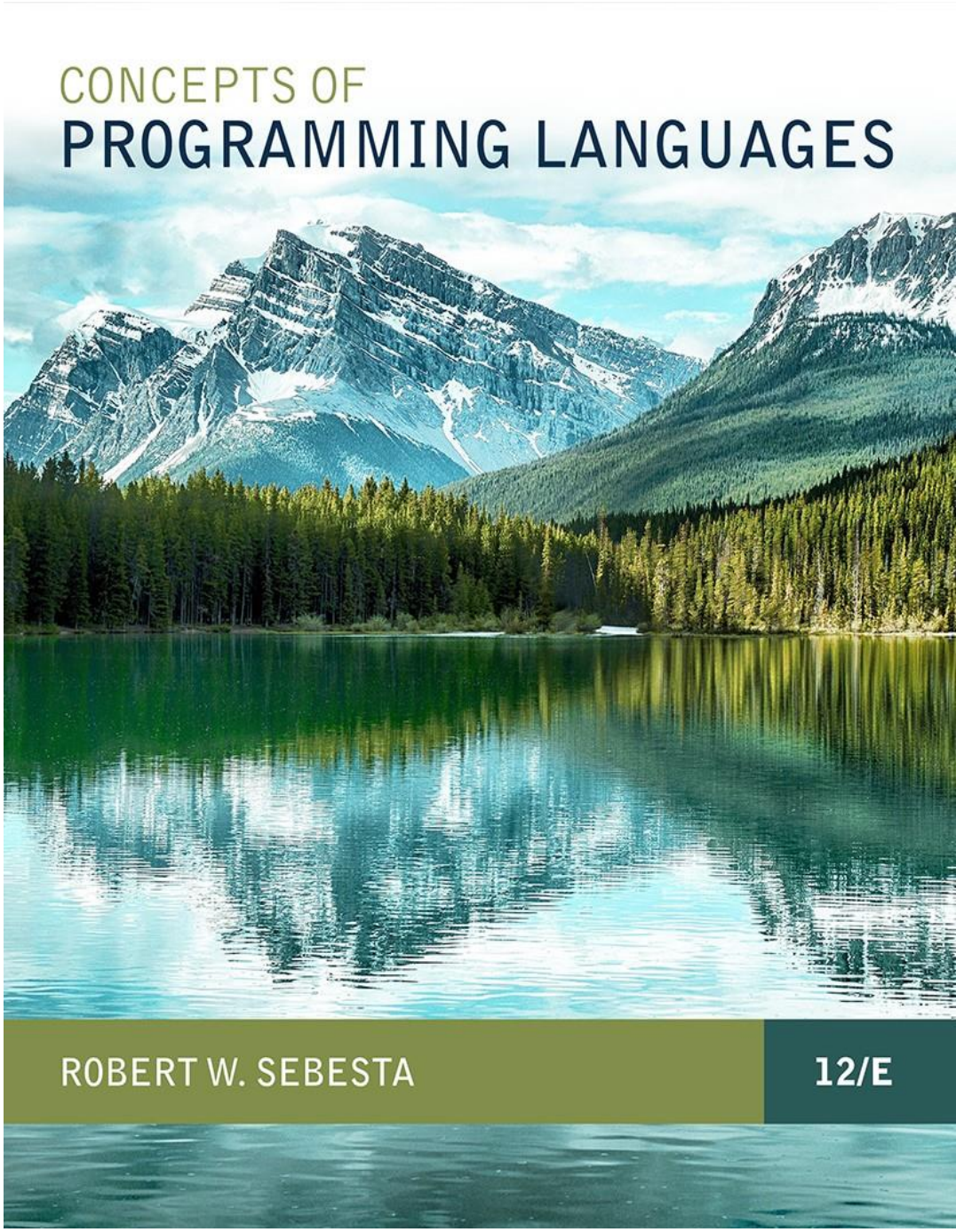
Chapter 1

기본적인 사항



ISBN 0-321-49362-1

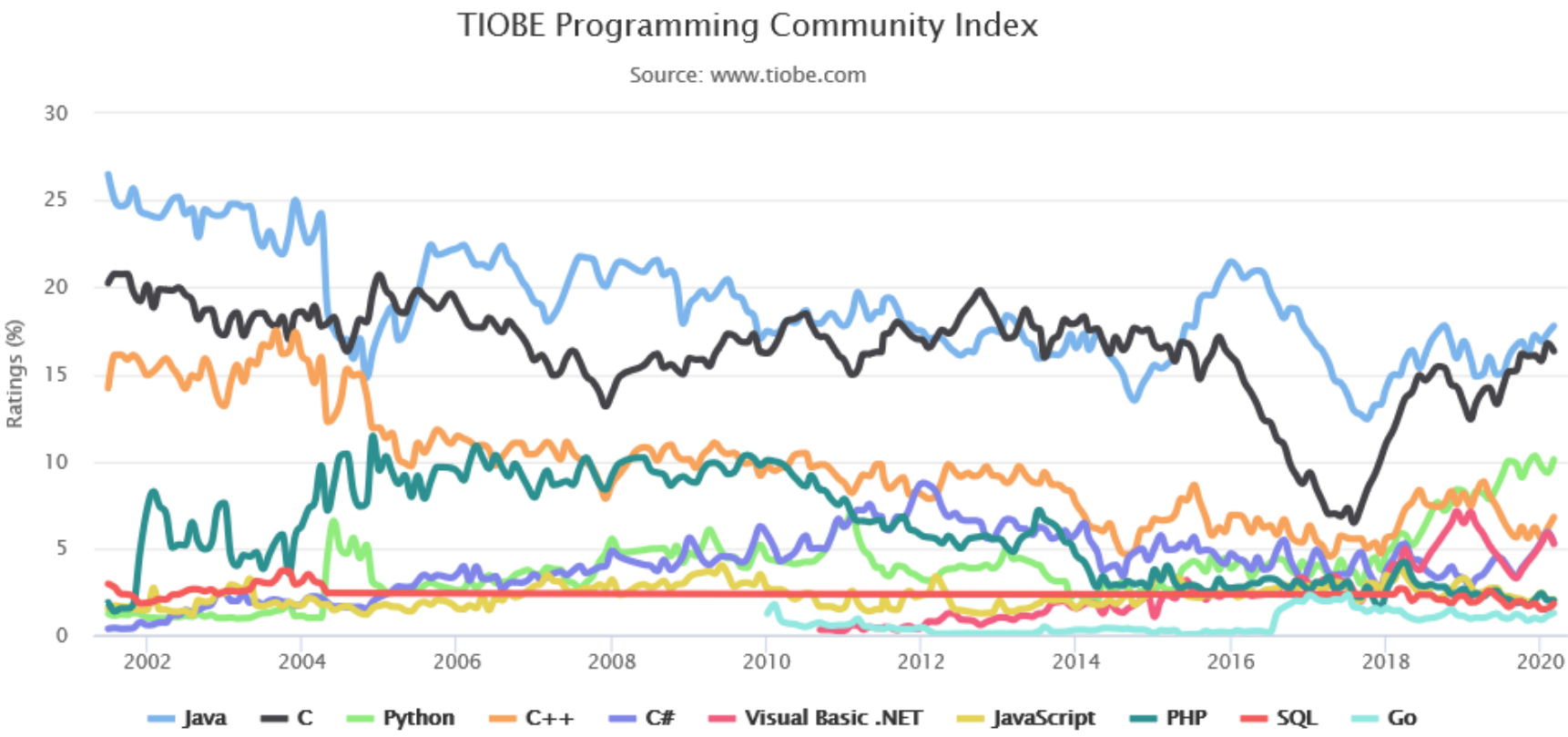
Chapter 1 Topics

- 프로그래밍 언어 사용 지수
- 프로그래밍 영역
- 언어 평가 기준
- 언어 설계에 미친 영향
- 언어 부류
- 언어 구현 방법
- 주요 프로그래밍언어

프로그래밍언어의 사용 지수(TIOBE 인덱스)

: 2020년 3월 기준(http://www.tiobe.com/tiobe_index/index.htm)

Mar 2020	Mar 2019	Change	Programming Language	Ratings	Change
1	1		Java	17.78%	+2.90%
2	2		C	16.33%	+3.03%
3	3		Python	10.11%	+1.85%
4	4		C++	6.79%	-1.34%
5	6	↑	C#	5.32%	+2.05%
6	5	↓	Visual Basic .NET	5.26%	-1.17%
7	7		JavaScript	2.05%	-0.38%
8	8		PHP	2.02%	-0.40%
9	9		SQL	1.83%	-0.09%
10	18	⇈	Go	1.28%	+0.26%
11	14	↑	R	1.26%	-0.02%
12	12		Assembly language	1.25%	-0.16%
13	17	⇈	Swift	1.24%	+0.08%
14	15	↑	Ruby	1.05%	-0.15%
15	11	⇊	MATLAB	0.99%	-0.48%
16	22	⇈	PL/SQL	0.98%	+0.25%
17	13	⇊	Perl	0.91%	-0.40%
18	20	↑	Visual Basic	0.77%	-0.19%
19	10	⇊	Objective-C	0.73%	-0.95%
20	19	↓	Delphi/Object Pascal	0.71%	-0.30%


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Java
C
Python
C++
C#
Visual Basic .NET
JavaScript
PHP
SQL
Go

Position	Programming Language	Ratings
21	SAS	0.70%
22	Scratch	0.69%
23	D	0.68%
24	Dart	0.60%
25	Transact-SQL	0.53%
26	COBOL	0.48%
27	ABAP	0.45%
28	Scala	0.42%
29	Lisp	0.37%
30	Rust	0.37%
31	Kotlin	0.35%
32	F#	0.32%
33	Fortran	0.29%
34	Lua	0.28%
35	PowerShell	0.28%
36	Groovy	0.25%
37	Ada	0.23%
38	Logo	0.23%
39	Haskell	0.22%
40	ML	0.20%

프로그래밍 영역

- Scientific applications
  - Large numbers of floating point computations; use of arrays
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (ex, HTML), scripting (ex, PHP), general-purpose (ex, Java)

언어 평가 기준

- 판독성(**Readability**): the ease with which programs can be read and understood
- 작성력(**Writability**): the ease with which a language can be used to create programs
- 신뢰성(**Reliability**): conformance to specifications (i.e., performs to its specifications)
- 비용(**Cost**): the ultimate total cost

언어 평가 기준

- 언어 평가기준과 이 기준에 영향을 미치는 언어 특성

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

언어 평가 기준 : 판독성

- 전반적인 단순성(Overall simplicity)
  - 전체 기본구조와 특징은 습득 가능한 정도
  - 최소한의 특징 다중성(feature multiplicity)
    (ex) count=count+1;
  - 최소한의 연산자 중복
- 직교성(Orthogonality)
  - 언어의 기본구조들과 몇가지 연산이 조합되어 제어구조와 데이터구조가 생성됨
    (ex) 4개의 기본 데이터타입(int, float, double, character) 과
    2개의 타입 연산자(array, pointer)의 조합
  - 모든 가능한 조합이 유효; 직교성 결여 => 예외사항 존재
    (ex) 함수반환값의 자료형(C언어, Algol68)
- 데이터 타입
  - 적절한 데이터 타입 제공 ⇒ 판독성 ↑
    (ex) timeout=true;
- 구문 설계 선택사항

언어 평가 기준 : 판독성

- 구문 설계 선택사항
  - 식별자의 형식 : 유연한 구성

    (ex) Fotran95에서 Do, End도 식별자 가능
  - 복합문을 구성하는 방법

    (ex) while, class, for에서 'end' 나 '}'의 범위 결정
    - ✓ Ada는 end if, end loop 사용 ⇒ 많은 예약어, 판독성↑
    - ✓ C++는 '}' 사용 ⇒적은 예약어, 단순성↑
  - 형식과 의미 : 키워드, 형식으로부터 그 의미가 파악되도록 설계

    (ex) static이 함수 내부, 외부에 쓰였을 때 의미가 다름

언어 평가 기준 : 작성력

- 단순성과 직교성
  - 적은 수의 기본구조와 이들을 조합하기 위한 일관된 규칙을 갖는 것 > 많은 기본 구조를 갖는 것
  - 과도한 직교성은 작성력에 방해(프로그램 오류 탐지 힘듦)

- 추상화의 지원
  - 세부사항을 무시하고 복잡한 구조나 연산을 정의하고 사용하는 방법
  - 프로세스 추상화, 데이터 추상화

- 표현력
  - 많은 양의 계산을 매우 작은 코드로 수행하게 하는 강력한 연산자 제공
    (ex) **count++**
  - 미리 정의된 함수와 강력한 연산자

언어 평가 기준 : 신뢰성

- 타입 검사
  - 실행시간 타입검사 비용 > 컴파일시간 타입검사 비용
- 예외처리
  - 프로그램이 실행시간 오류를 가로채서 이를 올바르게 처리한 후에 계속 실행하게 하는 기능
- 별칭
  - 동일한 기억장소를 접근하는 데 두 개 이상의 참조 방법을 제공 여부
    (ex) 동일한 변수를 가리키는 두 개의 포인터
- 판독성과 작성력
  - 부자연스러운 방법으로 알고리즘을 표현하는 언어는 작성된 프로그램도 신뢰성이 떨어질 가능성이 많음
  - 구현과 유지보수단계에서 판독성은 신뢰성에 영향을 미침

언어 평가 기준 : 비용

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Executing programs
- Reliability: poor reliability leads to high costs
- Maintaining programs

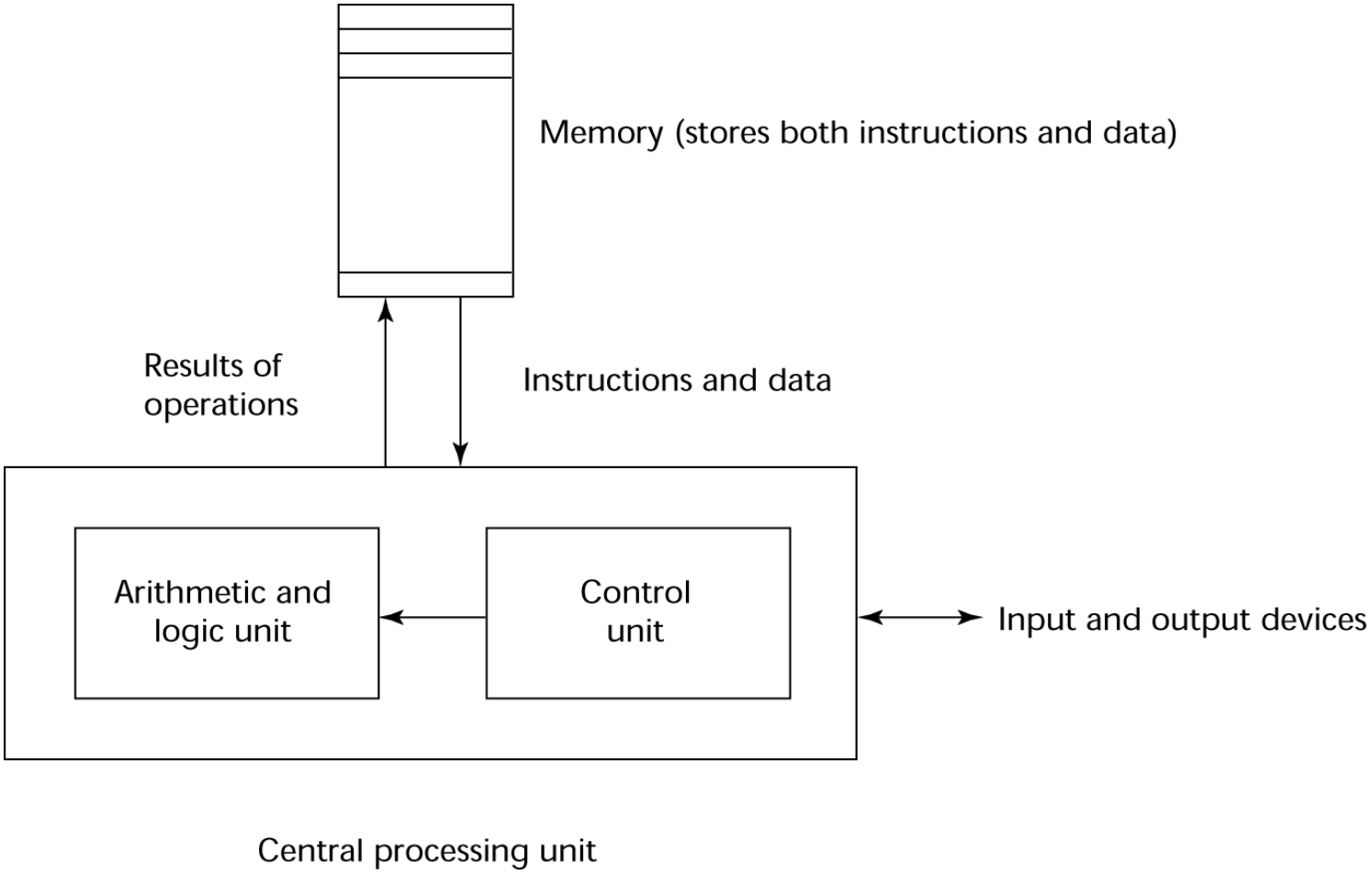
언어 설계에 미친 영향

- 컴퓨터 구조
  - 폰 노이만 구조 기반의 언어 개발
- 프로그램 설계 방법론
  - 새로운 SW 개발 방법론(eg. OOP)은 새로운 프로그래밍 패러다임과 새로운 프로그래밍언어를 유도

컴퓨터 구조의 영향

- 잘 알려진 컴퓨터 구조 : 폰 노이만 구조
- 대부분 명령형 언어
  - 데이터와 프로그램이 메모리에 저장
  - 메모리는 CPU와 분리
  - 명령어와 데이터는 메모리로부터 CPU에 전달
  - 명령형 언어의 기본
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

The von Neumann Architecture



The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

프로그래밍 방법론의 영향

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

언어 범주(Language Categories)

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages

    (Ex) C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
  - Main means of making computations is by applying functions to given parameters

    (Ex) LISP, Scheme, ML, F#
- Logic
  - Rule-based (rules are specified in no particular order)

    (Ex) Prolog
- Markup/programming hybrid
  - Markup languages extended to support some programming

    (Ex) JSTL, XSLT

언어구현방법(Implementation Methods)

- Compilation
  - Programs are translated into machine language; includes JIT systems
  - Use: Large commercial applications

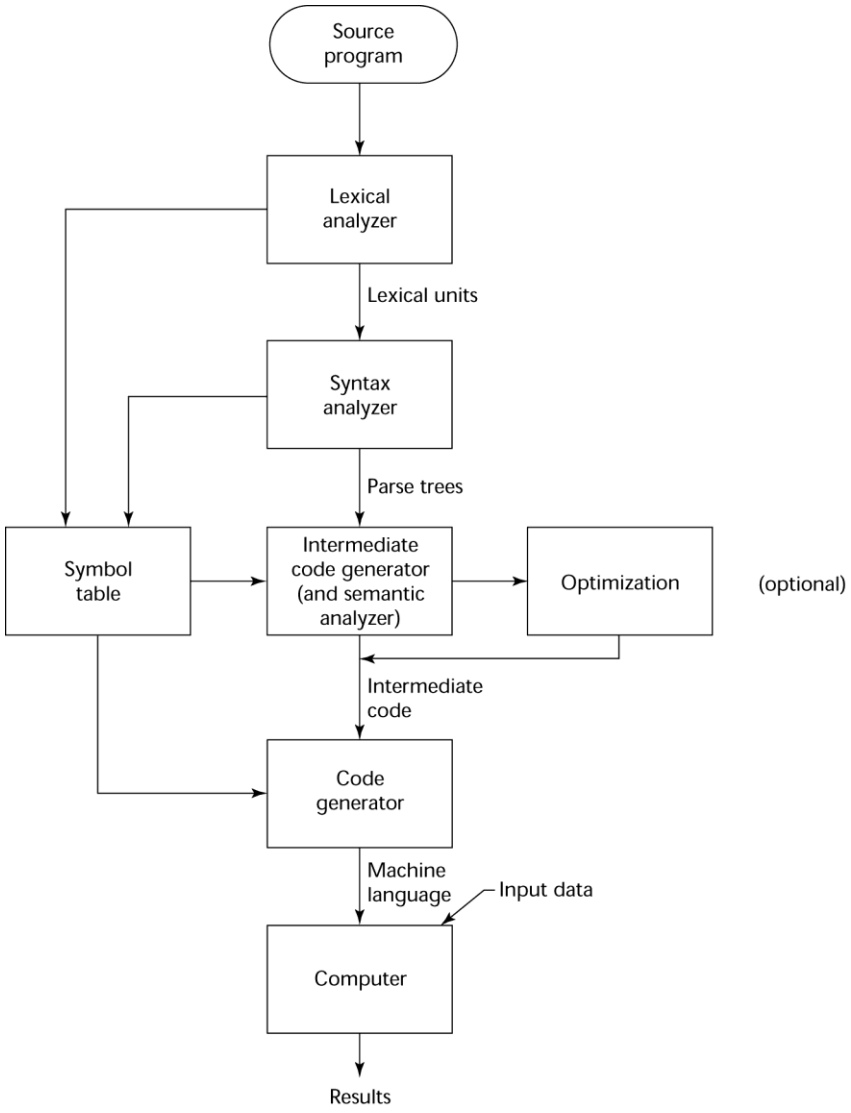
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue

- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters
  - Use: Small and medium systems when efficiency is not the first concern

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

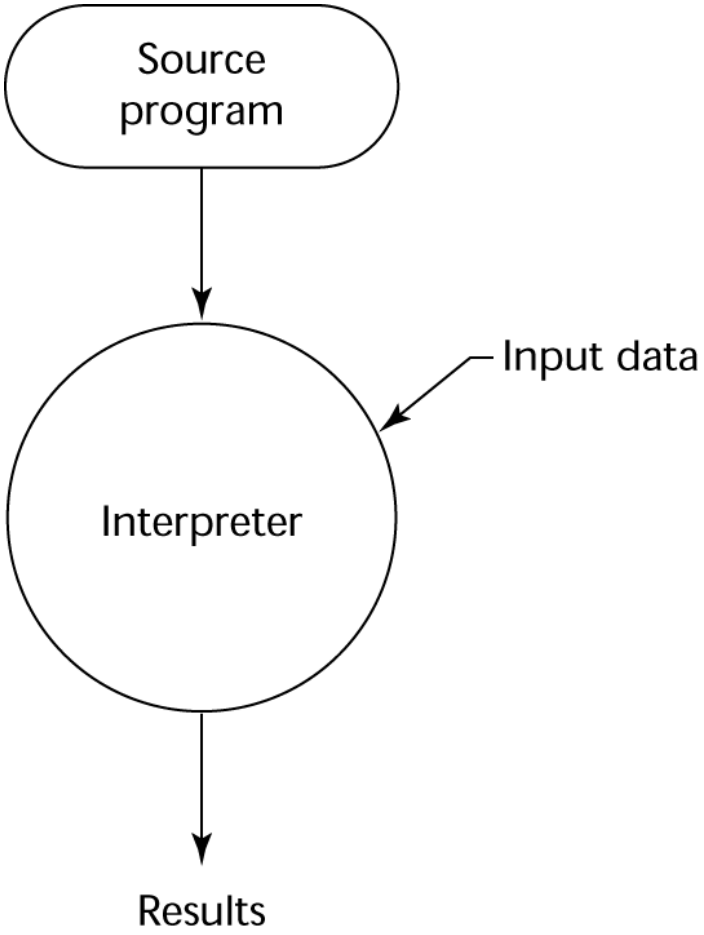
The Compilation Process



Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

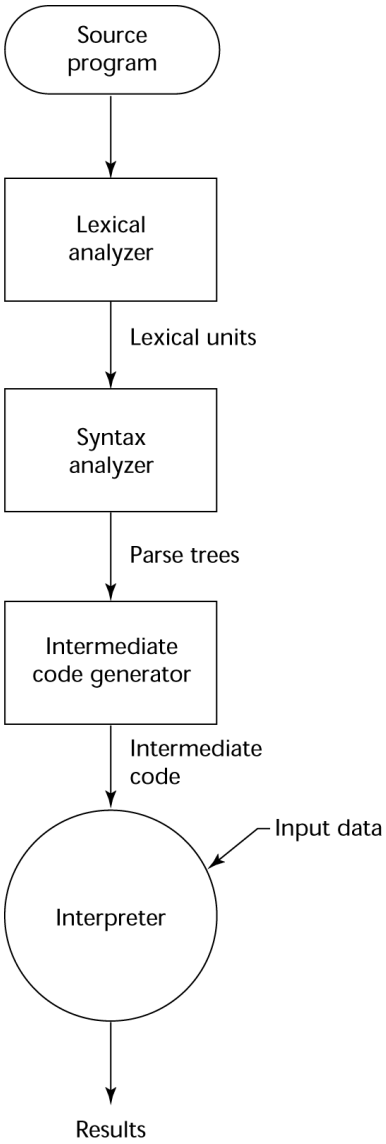
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Compiled vs. interpreted

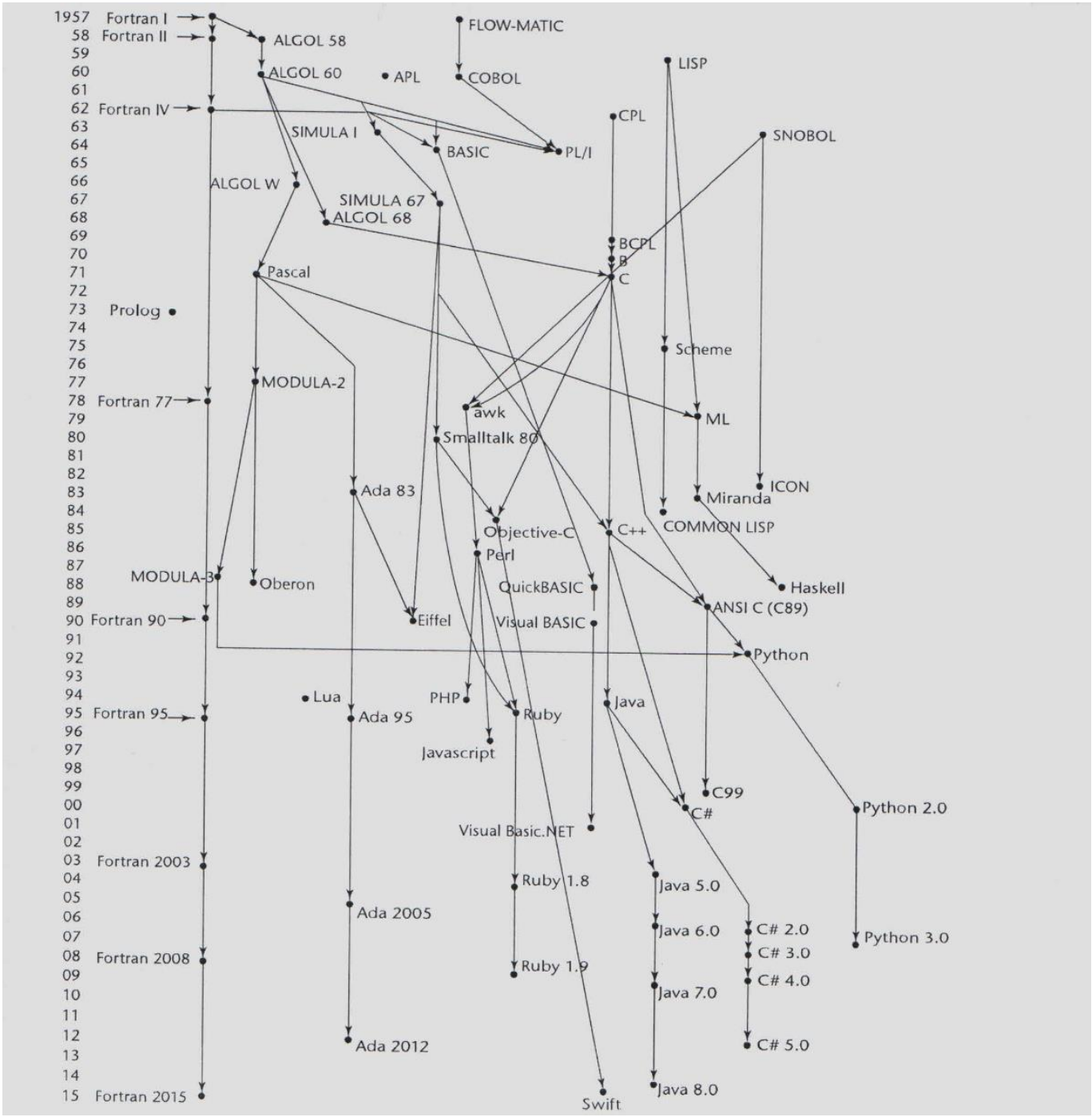
- Compiled
  - Static typing
  - Machine dependent
  - Error checking by compiler
  - Efficiency
- Interpreted
  - Dynamic typing
  - Portability
  - Run-time error checking
  - Flexibility
- In many languages we are often somewhere between!

주요 프로그래밍언어

- 1957 Fortran (Formula Translator)
- 1958 LISP (LISt Processor)
- 1959 COBOL (Common Business Oriented Language)
- 1960 ALGOL 60 (Algorithmic Language)
- 1964 BASIC (Beginner's All-purpose Symbolic Instruction Code)
- 1967 Simula (first object-oriented lang.)
- 1970 Pascal, Forth
- 1972 C, Prolog, Smalltalk
- 1975 Scheme (Lisp + Algol)
- 1978 ML (Meta-Language)
- 1980 Ada

주요 프로그래밍언어

- 1983 C++, Objective-C
- 1984 Common Lisp (Lisp + OO)
- 1986 Erlang
- 1987 Perl
- 1990 Haskell
- 1991 Python
- 1995 Java, JavaScript, Ruby, PHP
- 2001 C#
- 2002 F#
- 2003 Scala
- 2007 Clojure
- 2009 Go; '11 Dart, '12 Rust, '14 Swift ...


1957 Fortran I
58 Fortran II
59
60
61
62 Fortran IV
63
64
65
66
67
68
69
70
71
72
73 Prolog
74
75
76
77
78 Fortran 77
79
80
81
82
83
84
85
86
87
88
89
90 Fortran 90
91
92
93
94
95 Fortran 95
96
97
98
99
00
01
02
03 Fortran 2003
04
05
06
07
08 Fortran 2008
09
10
11
12
13
14
15 Fortran 2015
ALGOL 58
ALGOL 60
FLOW-MATIC
APL
COBOL
LISP
CPL
SNOBOL
SIMULA I
BASIC
PL/I
ALGOL W
SIMULA 67
ALGOL 68
BCPL
B
C
Pascal
Scheme
MODULA-2
awk
ML
Smalltalk 80
ICON
Ada 83
Miranda
Objective-C
C++
COMMON LISP
Perl
MODULA-3
Oberon
QuickBASIC
Haskell
ANSI C (C89)
Eiffel
Visual BASIC
Python
Lua
PHP
Java
Ada 95
Ruby
Javascript
C99
C#
Python 2.0
Visual Basic.NET
Ruby 1.8
Java 5.0
Ada 2005
Java 6.0
C# 2.0
Python 3.0
C# 3.0
Ruby 1.9
C# 4.0
Java 7.0
Ada 2012
C# 5.0
Swift
Java 8.0