

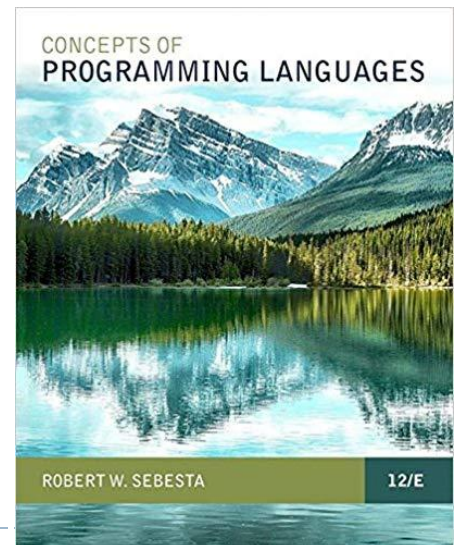
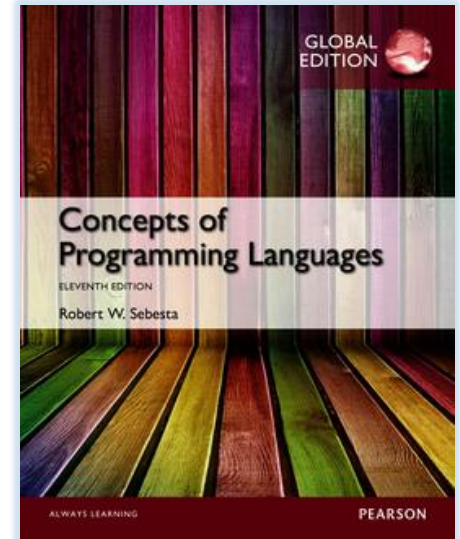
Course Information

▶ Staff

- ▶ ABUHMED, Tamer (타메르)
(tamer@skku.edu)
- ▶ Yoon Si-hyung 윤시형
- ▶ Lee Jae-bin 이재빈

▶ Textbook

- ▶ Concepts of Programming Languages,
11/12th Edition, Robert W. Sebesta,
Pearson, 2015/2018.



Course Overview

▶ Course Description

- ▶ The course covers the fundamental concepts of programming languages by discussing the design issues of the various language constructs and examining the design choices for these constructs in some of the most common languages.

▶ Grading Policy

- ▶ Midterm(20%) - Final exam(20%)
- ▶ Attendance (10%) - Assignments (50%)

▶ Cheating Policy

- ▶ Automatic F for both

“Programming languages are our most basic tools, and we must thoroughly master them to use them effectively”

Programming Language Design Concepts by **David A. Watt** and **William Findlay**

Prerequisites for the course

- ▶ Good knowledge of at least one programming language (C++, Java, etc.)
- ▶ Experience in writing programs (larger than simple toy programs)
- ▶ Knowledge of basic computer science terms (recursion, algorithms, memory management, functions, parameters, objects, inheritance, modularity)

Some topics of the course

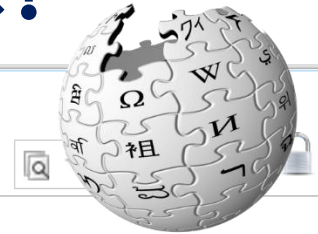
- ▶ Programming paradigms
 - ▶ Imperative, Functional, Logic, Script
- ▶ General syntax and semantics of languages
- ▶ Phases of program compilation and interpretation
- ▶ Variable lifetimes, binding and memory management
- ▶ Typing and type systems
- ▶ Subprograms and parameter passing mechanisms
- ▶ Compromises in implementing language features
- ▶ Generic programming, concurrency features,...

Topics of the lecture

- ▶ Why study PL concepts?
- ▶ Programming domains
- ▶ PL evaluation criteria
- ▶ What influences PL design?
- ▶ Tradeoffs faced by programming languages
- ▶ Implementation methods
- ▶ Programming environments

Introduction

What is a programming language?



Programming language

From Wikipedia, the free encyclopedia

A **programming language** is a formal constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

The earliest programming languages preceded the invention of the digital computer and were used to direct the behavior of machines such as Jacquard looms and player pianos.^[1] Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform), while other languages utilize other forms of program specification such as the declarative form (i.e. the desired result is specified, not how to achieve it).

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages (such as Perl) have a dominant implementation that is treated as a reference.

```
1 // class declaration
2 public class ProgrammingExample {
3
4     // method declaration
5     public void sayHello() {
6
7         // method output
8         System.out.println("Hello World!");
9     }
10 }
```

An example of source code written in the Java programming language, which will print the message "Hello World!" to the standard output when it is compiled and then run by the Java Virtual Machine.

Attempts to define a Programming Language

- ▶ A tool for humans to tell the computer what to do
- ▶ A means of formal and exact communication between humans
- ▶ A way of describing an abstract machine
- ▶ A way of describing computation that is readable by both man and machine



Where are we now?

```
01 .MODEL SMALL
02 .STACK 100H
03 .CODE
04
05 MOV AX, 0x3C
06 MOV BX, 0000000000001010B
07 ADD AX, BX
08 MOV BX, 14
09 SUB AX, BX
10
11 MOV AH, 04CH
12 INT 21H
```

Assembly code
1950s -1960s

```
10 INPUT "What is your name: ", US$
20 PRINT "Hello "; US$
30 INPUT "How many stars do you want: ", N
40 SS = ""
50 FOR I = 1 TO N
60 SS = SS + "*"
70 NEXT I
80 PRINT SS
90 INPUT "Do you want more stars? ", AS$
100 IF LEN(AS) = 0 THEN GOTO 90
110 AS$ = LEFT$(AS, 1)
120 IF AS$ = "Y" OR AS$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; US$
140 END
```

BASIC code
1960s -1970s

```
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int arr[] = { 1,2,7,12,9 };
8     int even=0;
9     int odd=0;
10    int length = sizeof(arr)/sizeof(arr[0]);
11
12    for(int i=0;i<length;i++)
13    {
14        if(arr[i]%2==0)
15        {
16            even++;
17        }
18        else
19        {
20            odd++;
21        }
22    }
23
24    cout<<"Even numbers " <<even<<endl;
25    cout<<"Odd numbers " <<odd;
26    return 0;
27 }
```

C & C++
1970s - present

```
1 import java.util.*;
2 class matrix{
3     public static void main (String args[]) {
4         Scanner sc = new Scanner (System.in);
5         int num[][]=new int[3][3];
6         for (int i=0;i<num.length;i++) {
7             for (int j=0;j<num.length;j++) {
8                 num[i][j]=sc.nextInt();
9             }
10        }
11
12        for (int i=0;i<num.length;i++) {
13            for (int j=0;j<num.length;j++) {
14                System.out.print (num[i][j]+" ");
15            }
16            System.out.println();
17        }
18    }
19 }
```

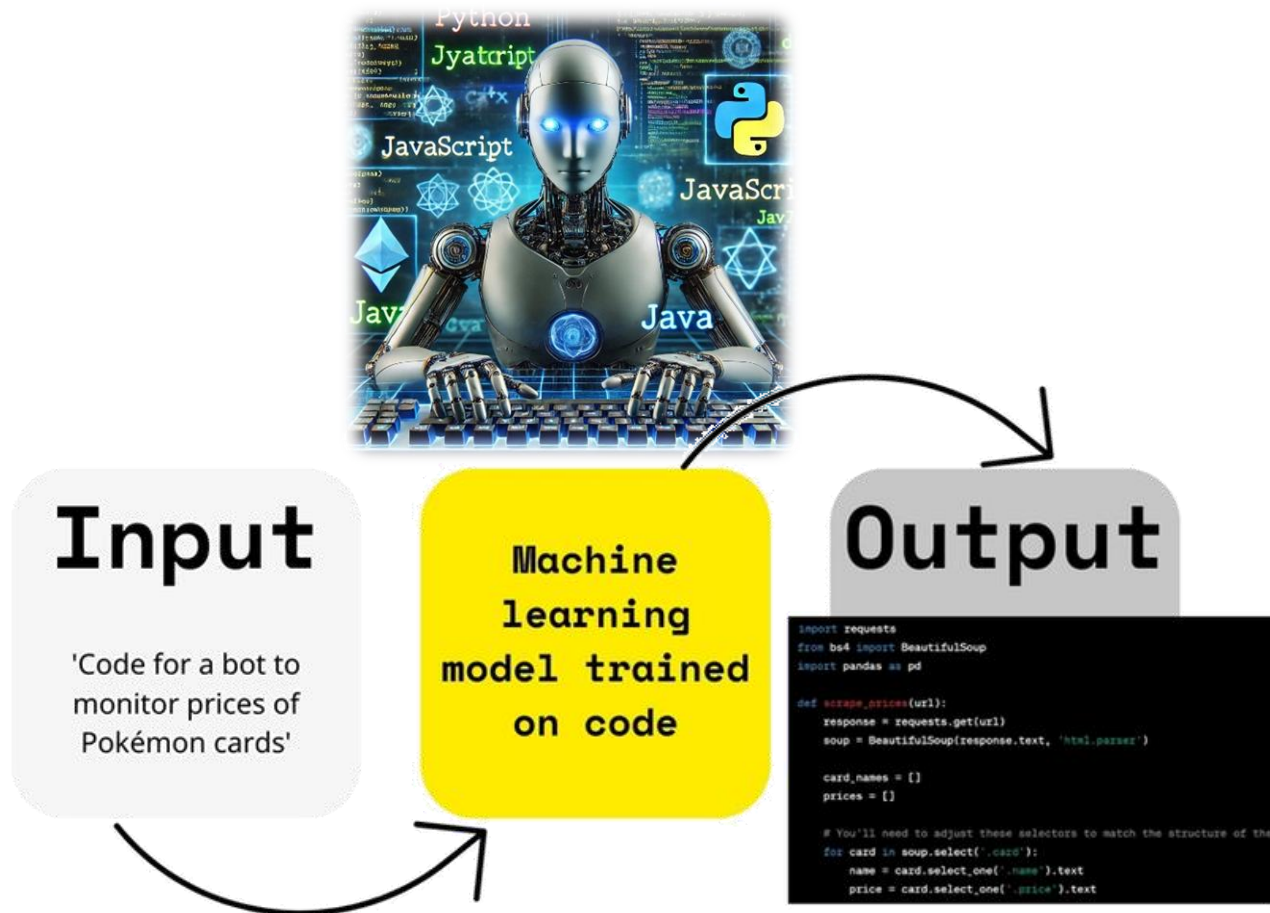
Java
1995 -present

```
1 #modeling the person class
2 class Person():
3     #method to initialize name and age attributes.
4     def __init__(self,name, age):
5         self.name = name
6         self.age = age
7     #method to demonstrate what a person eats
8     def eat(self):
9         print(self.name.title() + "eats Matooke and rice")
10        print("She is"+ str(self.age) + " years old")
11    def drink(self):
12        print("Drinks water")
13
14    #instantiating a class.
15    my_sister = Person("Haniifa", 30)
16    #Accessing the class method through the class object.
17    my_sister.eat()
```

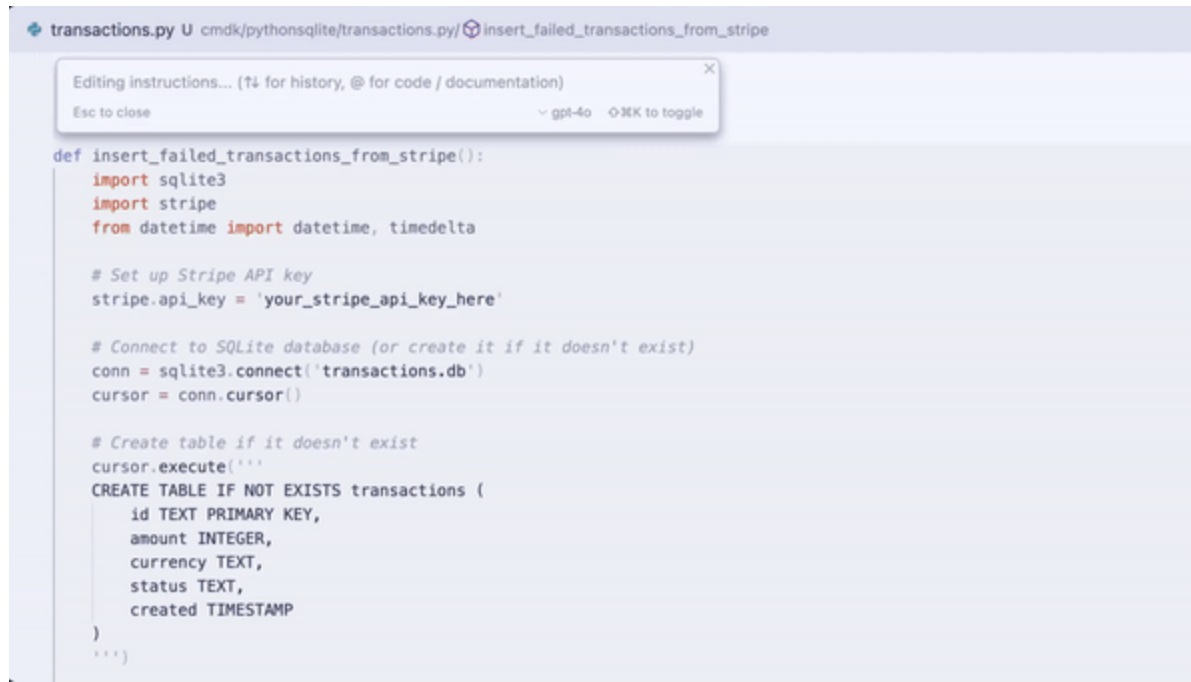
Python
2000 -present



What about the AI area?



Do we still need to learn programming languages?



```
transactions.py U cmdk/pythonsqlite/transactions.py/ insert_failed_transactions_from_stripe

Editing instructions... (↑↓ for history, @ for code / documentation)
Esc to close ~ gpt-4o ⇌ KK to toggle

def insert_failed_transactions_from_stripe():
    import sqlite3
    import stripe
    from datetime import datetime, timedelta

    # Set up Stripe API key
    stripe.api_key = 'your_stripe_api_key_here'

    # Connect to SQLite database (or create it if it doesn't exist)
    conn = sqlite3.connect('transactions.db')
    cursor = conn.cursor()

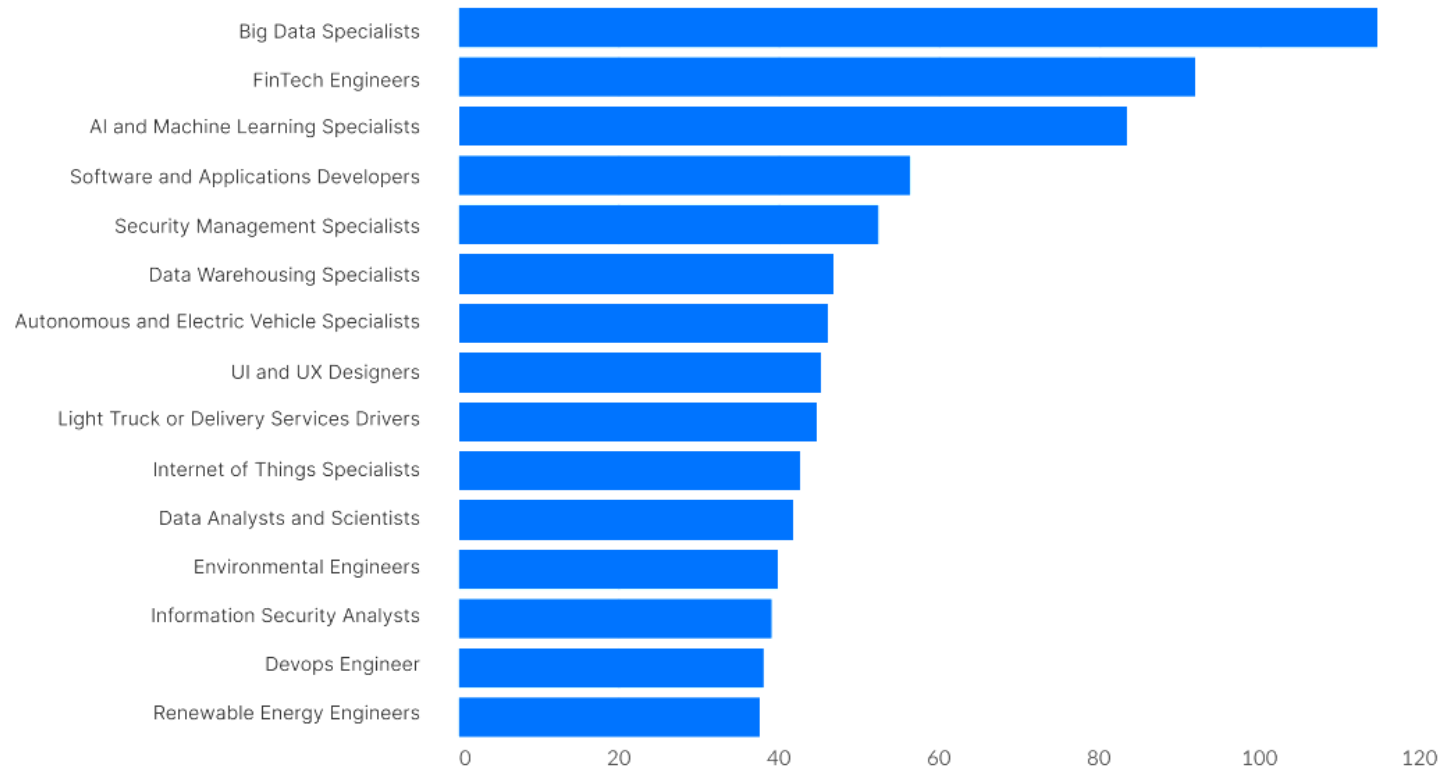
    # Create table if it doesn't exist
    cursor.execute('''
CREATE TABLE IF NOT EXISTS transactions (
    id TEXT PRIMARY KEY,
    amount INTEGER,
    currency TEXT,
    status TEXT,
    created TIMESTAMP
)
''')
```

Do we still need to learn programming languages?

Short answer: Definitely, Yes

Do we still need to learn programming languages?

Top fastest growing jobs, 2025-2030

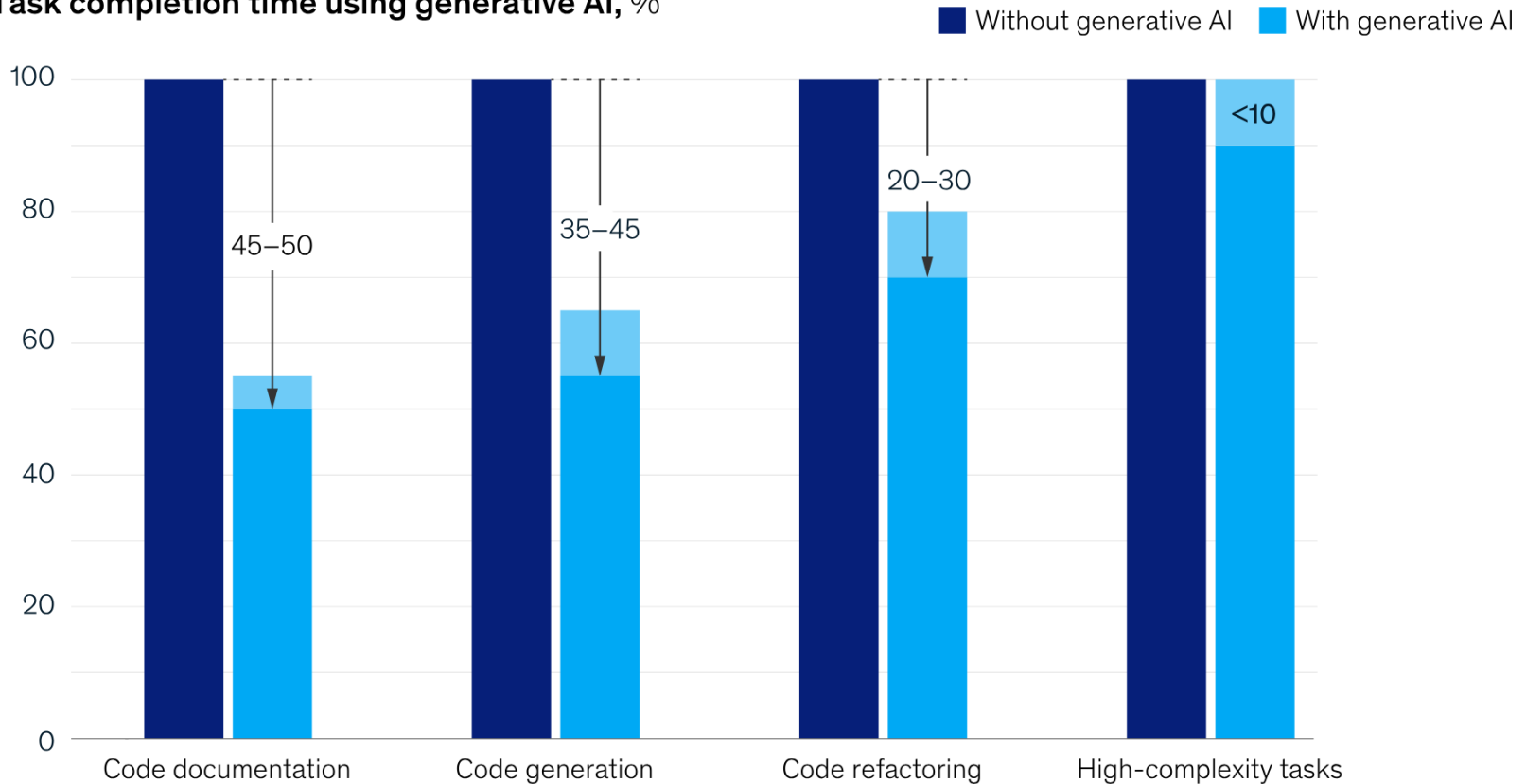


Source: World Economic Forum, Future of Jobs Report 2025

Where generative AI shined

Generative AI can increase developer speed, but less so for complex tasks.

Task completion time using generative AI, %

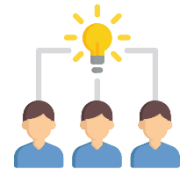


McKinsey & Company

Repetitive work, first draft of new code, Accelerating updates,
Increasing developers' ability to tackle new challenges

Why programming languages?

- ▶ Reasons to study theory programming languages
 - ▶ Enhanced ability to learn new languages
 - ▶ Borrowing ideas from other languages
 - ▶ Ability to express ideas and programming concepts.
 - ▶ **Choice** of programming language
 - ▶ Understanding the implementation behind languages
 - ▶ Mastering different programming paradigms
 - ▶ Designing your own "programming language"



Programming Language Properties

- ▶ Programming language must be **universal**.
 - ▶ Every problem must have a programming solution.
 - ▶ Any language in which we can define recursive functions is universal.
- ▶ Reasonably *natural* for solving problems within its intended application area.
- ▶ **Implementable** on a computer.
 - ▶ Possible to execute every well-formed program in the language.
 - ▶ Mathematical notation is not implementable (it is possible to formulate problems that cannot be solved in any computer).
- ▶ Acceptably efficient implementation

Language Evaluation Criteria

- ▶ **Readability:** programs can be read and understood
- ▶ **Writability:** language can be used easily to create programs
- ▶ **Reliability:** performs to its specifications
- ▶ **Cost:** the ultimate total cost
- ▶ **Portability:** moved from one to another
- ▶ **Generality:** wide range of applications



Evaluation Criteria: Readability

- ▶ How easy is it to read and understand programs written in the programming language?
- ▶ Arguably the most important criterion!
- ▶ Factors effecting readability include:
 - ▶ Overall simplicity
 - ▶ Too many features is bad as is a multiplicity of features
 - ▶ Orthogonality: a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language
 - ▶ Makes the language easy to learn and read
 - ▶ Meaning is context independent
 - ▶ Control statements
 - ▶ Data type and structures
 - ▶ Syntax considerations

Readability: simplicity

- ▶ Programming languages with a large number of basic components are harder to learn; most programmers using these languages tend to learn and use subsets of the whole language.
- ▶ Complex languages have multiplicity (more than one way to accomplish an operation).
- ▶ Overloading operators can reduce the clarity of the program's meaning

Readability: Orthogonality

- ▶ For a programming language to be orthogonal, language constructs should not behave differently in different contexts.
- ▶ The fact that Modula-2's constant expressions may not include function calls can be viewed as a nonorthogonality.
- ▶ Examples of Nonorthogonalities
 - ▶ In C/C++, arrays types cannot be returned from a function
 - ▶ In C, local variables must be at the beginning of a block.
 - ▶ C passes ALL parameters by value except arrays (passed by reference).

Data Types and Structures

- ▶ A more diverse set of data types and the ability of programmers to create their own increased program readability:
- ▶ Booleans make programs more readable:
`TimeOut = 1` vs. `TimeOut = True`
- ▶ The use of records to store complex data objects makes programs more readable:
`CHARACTER*30 NAME(100)`
`INTEGER AGE(100), EMPLOYEE_NUM(100)`
`REAL SALARY(100)`
 - ▶ Wouldn't it better if these were an array of records instead of 4 parallel arrays?

Syntax

- ▶ Most syntactic features in a programming language can enhance readability:
- ▶ Identifier forms – older languages (like FORTRAN) restrict the length of identifiers, which become less meaningful
- ▶ Special words – in addition to **while**, **do** and **for**, some languages use special words to close structures such as **endif** and **endwhile**. – Form and meaning
- ▶ In C a **static** variable within a function and outside a function mean two different things – this is undesirable.

Evaluation Criteria: Writability

How easy is it to write programs in the language?

Factors effecting writability:

- ▶ Simplicity and orthogonality
- ▶ Support for abstraction
- ▶ Expressivity: set of relatively convenient ways of specifying operations
- ▶ Fit for the domain and problem

Evaluation Criteria: Reliability

- ▶ **Type Checking** – a large factor in program reliability. Compile-time type checking is more desirable. C's lack of parameter type checking leads to many reliability problems.
- ▶ **Exception Handling** – the ability to catch runtime errors and make corrections can prevent reliability problems.
- ▶ **Aliasing** – having two or more ways of referencing the same data object can cause unnecessary errors.

Evaluation Criteria: Cost

- ▶ Categories:
 - ▶ Programmer training
 - ▶ Software creation
 - ▶ Compilation
 - ▶ Execution
 - ▶ Compiler cost
 - ▶ Poor reliability
 - ▶ Maintenance

Evaluation Criteria: others

- ▶ Portability
- ▶ Generality
- ▶ Well-definedness: The completeness and precision of the language's official definition
- ▶ Good fit for hardware (e.g., cell) or environment (e.g., Web)
- ▶ etc...

Recap: Language Evaluation Criteria

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

Language Design Trade-Offs



- ▶ **Reliability vs. cost of execution**

- ▶ Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

- ▶ **Readability vs. writability**

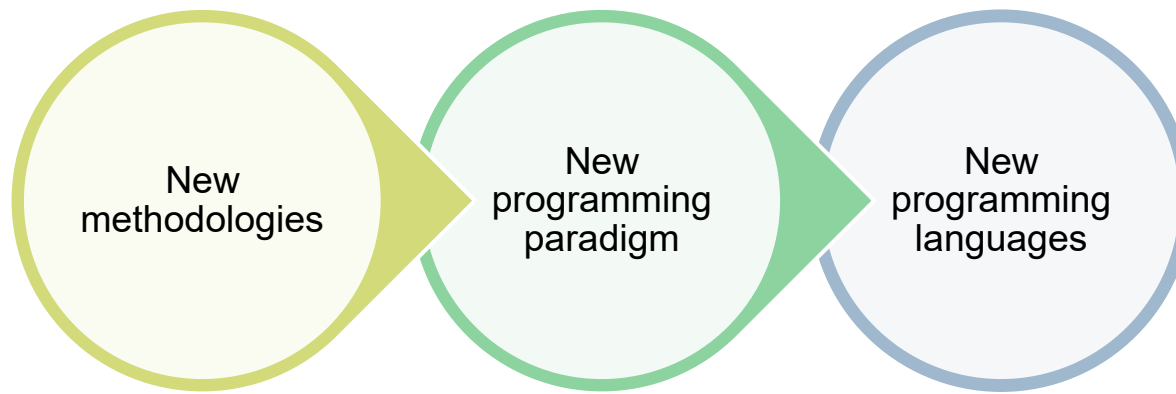
Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

- ▶ **Writability (flexibility) vs. reliability**

- ▶ Example: C++ pointers are powerful and very flexible but are unreliable

Influences on Language Design

- ▶ Computer Architecture
 - ▶ Languages are developed around the prevalent computer architecture, known as the **von Neumann** architecture
- ▶ Program Design Methodologies
 - ▶ New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages



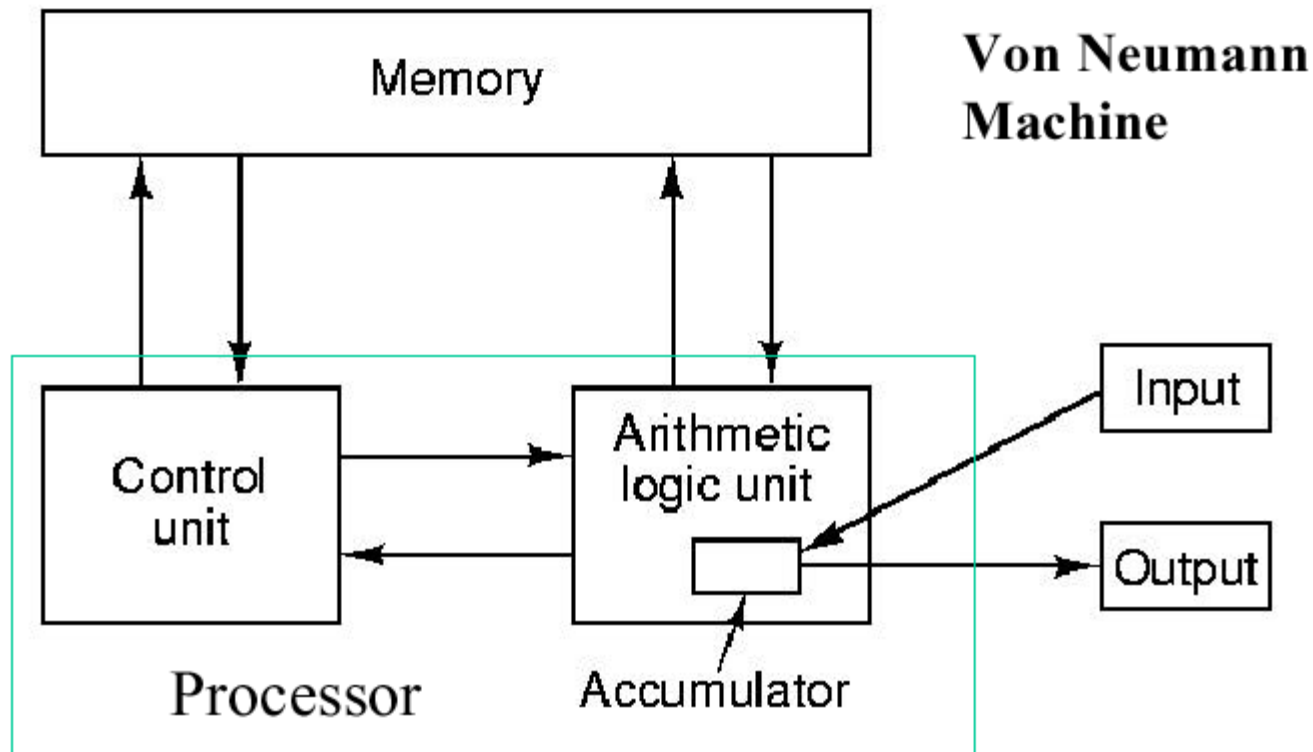
Language Design Influences

Computer architecture

- ▶ We use imperative languages, at least in part, because we use von Neumann machines
 - ▶ John von Neuman is generally considered to be the inventor of the "stored program" machines, the class to which most of today's computers belong
 - ▶ One CPU + one memory system that contains *both* program and data
- ▶ Focus on moving data and program instructions between registers in CPU to memory locations
- ▶ Fundamentally sequential




Von Neumann Architecture



Language Design Influences:

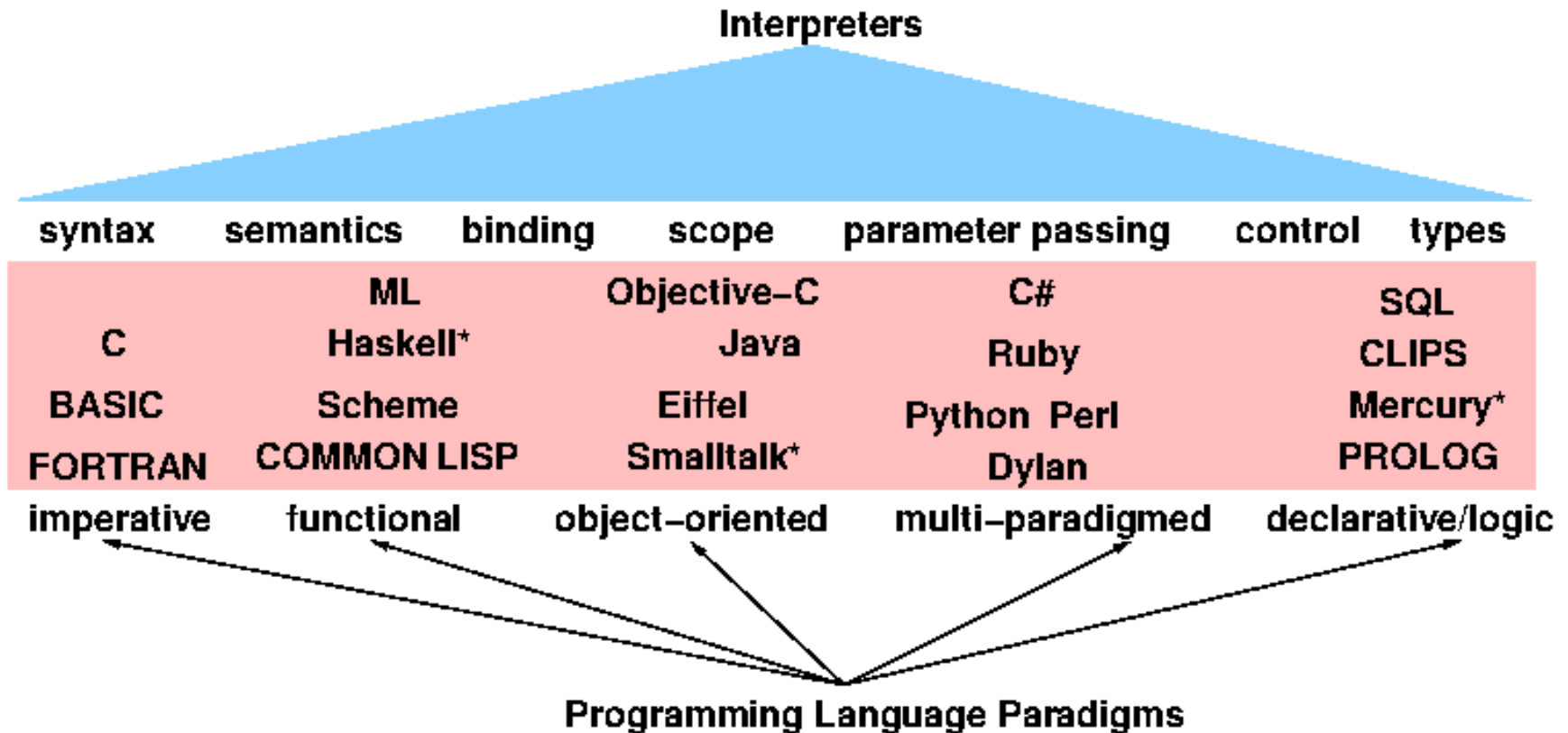
Programming methodologies

- 
- ▶ *50s and early 60s*: Simple applications; worry about machine efficiency
 - ▶ *Late 60s*: People efficiency became important; readability, better control structures. maintainability
 - ▶ *Late 70s*: Data abstraction
 - ▶ *Middle 80s*: Object-oriented programming
 - ▶ *90s*: distributed programs, Internet
 - ▶ *00s*: Web, user interfaces, graphics, mobile, services
 - ▶ *10s*: parallel computing, cloud computing?, pervasive computing?, semantic computing?, virtual machines?

Language Categories

- ▶ The big four PL paradigms:
 - ▶ Imperative or procedural (e.g., Fortran, C)
 - ▶ Object-oriented (e.g. Smalltalk, Java)
 - ▶ Functional (e.g., Lisp, ML)
 - ▶ Rule based (e.g. Prolog, Jess)
- ▶ Others:
 - ▶ Scripting (e.g., Python, Perl, PHP, Ruby)
 - ▶ Constraint (e.g., Eclipse)
 - ▶ Concurrent (Occam)
 - ▶ ...

Programming language paradigms



Implementation methods

- ▶ Direct execution by hardware

e.g., native machine language

- ▶ Compilation to another language

e.g., C compiled to native machine language

- ▶ Interpretation: direct execution by software

e.g., csh, Lisp (traditionally), Python, JavaScript

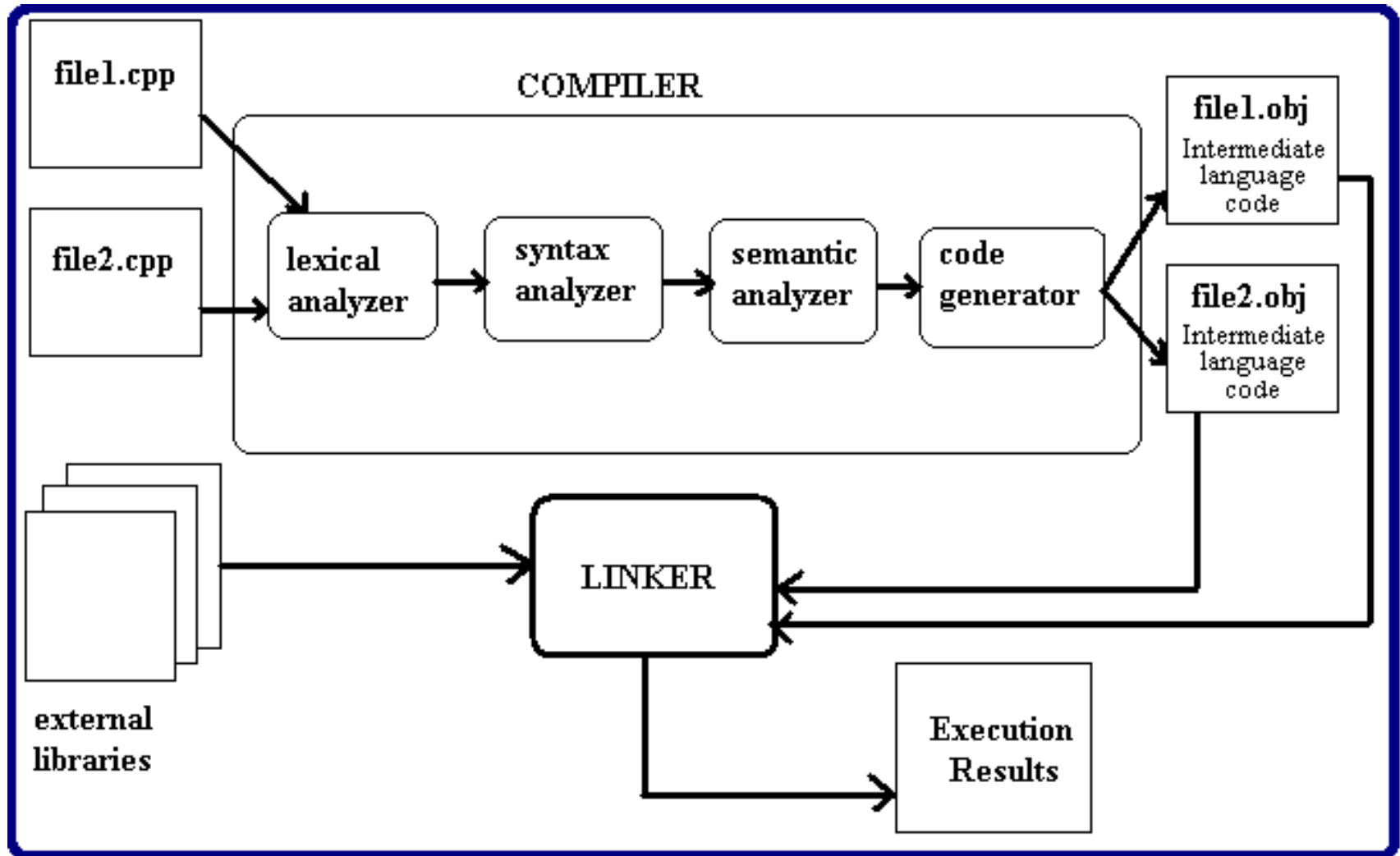
- ▶ Hybrid: compilation then interpretation

Compilation to another language (aka bytecode), then interpreted by a ‘virtual machine’, e.g., Java, Perl

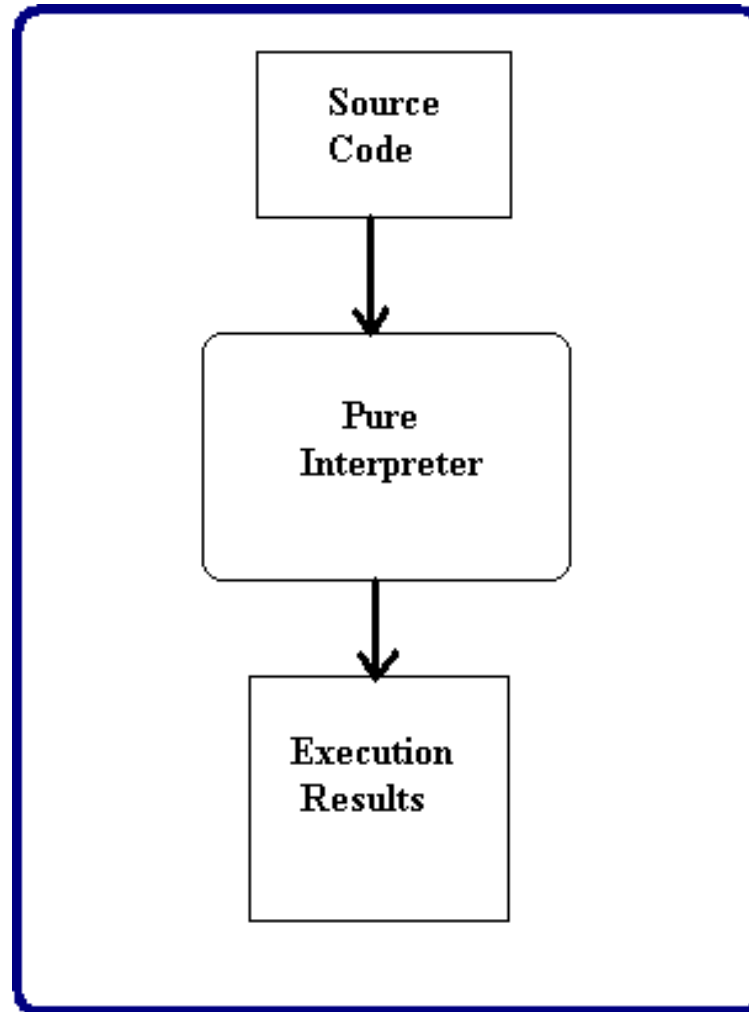
- ▶ Just-in-time compilation

Dynamically compile some bytecode to native code (e.g., V8 javascript engine)

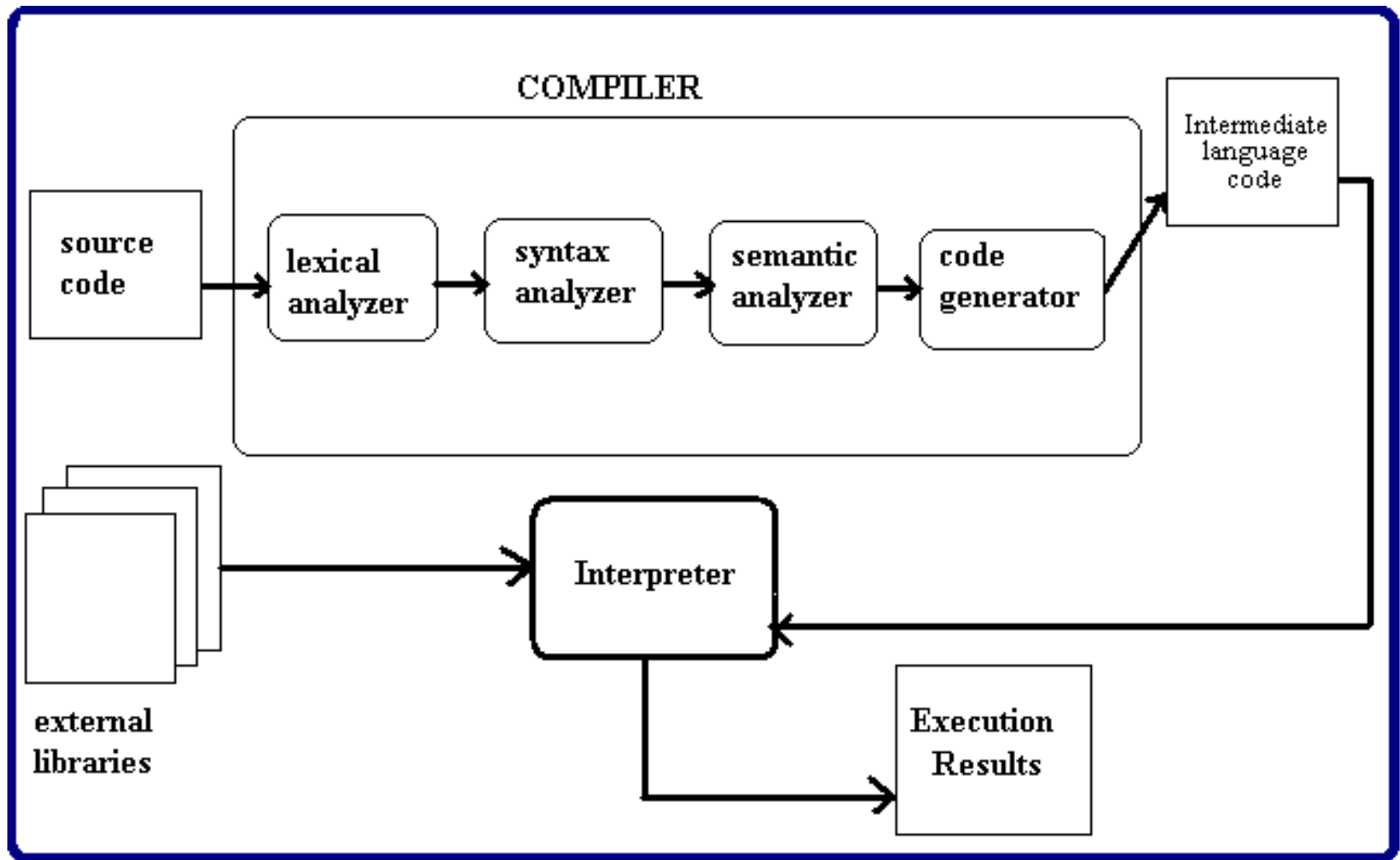
Compilation



Interpretation

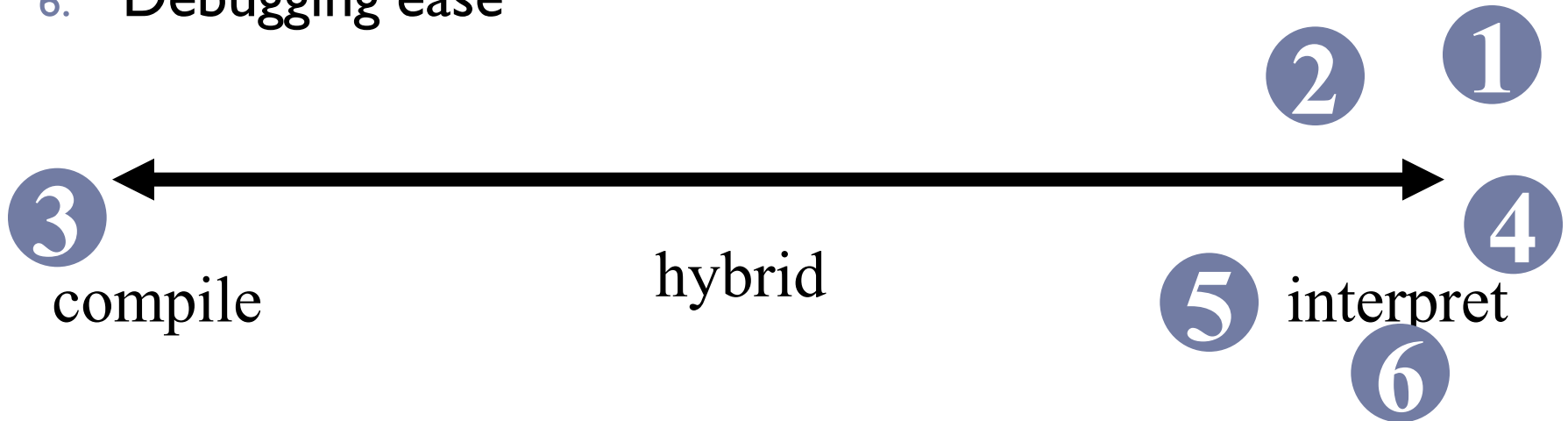


Hybrid



Implementation issues

1. Complexity of compiler/interpreter
2. Translation speed
3. Execution speed
4. Code portability
5. Code compactness
6. Debugging ease



Programming Environments

- ▶ The collection of tools used in software development, often including an integrated editor, debugger, compiler, collaboration tool, etc.
- ▶ Modern Integrated Development Environments (IDEs) tend to be language specific, allowing them to offer support at the level at which the programmer thinks.
- ▶ Examples:
 - ▶ UNIX -- Operating system with tool collection
 - ▶ EMACS – a highly programmable text editor
 - ▶ Smalltalk -- A language processor/environment
 - ▶ Microsoft Visual C++ -- A large, complex visual environment
 - ▶ Your favorite Java environment: BlueJ, Jbuilder, J++, ...
 - ▶ Generic: IBM's Eclipse

Summary

- ▶ Programming languages have many aspects and uses
- ▶ There are many reasons to study the concepts underlying programming languages
- ▶ There are several criteria for evaluating PLs
- ▶ Programming languages are constantly evolving
- ▶ Classic techniques for executing PLs are compilation and interpretation, with variations