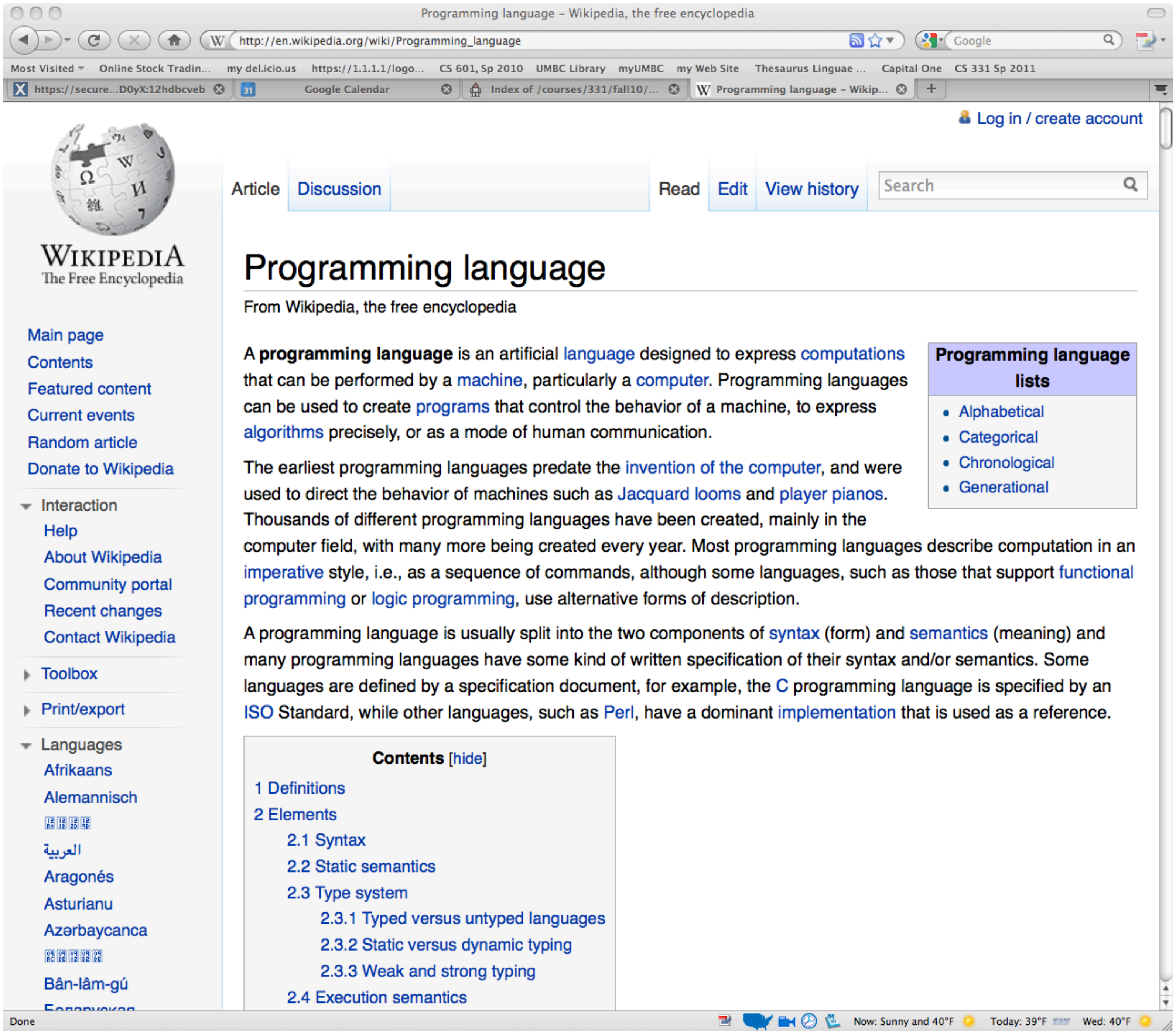
Programming Languages

Introduction

Overview

- Motivation
- Why study programming languages?
- Some key concepts

What is a programming language?



Programming language

From Wikipedia, the free encyclopedia

A **programming language** is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

A programming language is usually split into the two components of syntax (form) and semantics (meaning) and many programming languages have some kind of written specification of their syntax and/or semantics. 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 used as a reference.

Programming language lists

- Alphabetical
- Categorical
- Chronological
- Generational

Contents [hide]

What is a programming language?

"...there is no agreement on what a programming language really is and what its main purpose is supposed to be. Is a programming language a tool for instructing machines? A means of communicating between programmers? A vehicle for expressing high-level designs? A notation for algorithms? A way of expressing relationships between concepts? A tool for experimentation? A means of controlling computerized devices? My view is that a general-purpose programming language must be all of those to serve its diverse set of users. The only thing a language cannot be – and survive – is a mere collection of ''neat'' features."

-- Bjarne Stroustrup, The Design and Evolution of C++

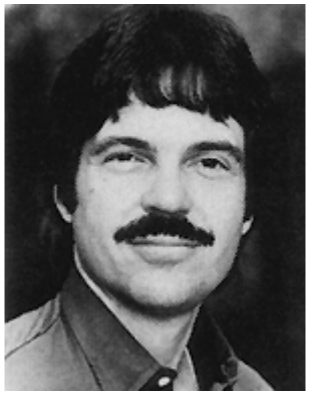
There is a link to the first chapter of this book dne_notes.pdf

On language and thought

- “The tools we use have a profound (and devious!) influence on our thinking habits, and therefore, on our thinking abilities.”
  -- Edsger Dijkstra,
  How do we tell truths that might hurt

- “A language that doesn't affect the way you think about programming, is not worth knowing”
  -- Alan Perlis

On languages and thought (2)



"What doesn't exist are really powerful general forms of arguing with computers right now. So we have to have special orders coming in on special cases and then think up ways to do it. Some of these are generalizable and eventually you will get an actual engineering discipline."

-- Alan Kay, Educom Review

Alan Kay is one of the inventors of the Smalltalk programming language and one of the fathers of the idea of OOP. He is the conceiver of the laptop computer and the architect of the modern windowing GUI.

Some General Underlying Issues

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

Why study Programming Language Concepts?

- Increased capacity to express programming concepts
- Improved background for choosing appropriate languages
- Enhanced ability to learn new languages
- Improved understanding of the significance of implementation
- Increased ability to design new languages
- Mastering different programming paradigms

Programming Domains

- Scientific applications
- Business applications
- Artificial intelligence
- Systems programming
- Scripting languages
- Special purpose languages

Language Evaluation Criteria

- Readability
- Writability
- Reliability
- Cost
- Etc…

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**

Evaluation Criteria: Writability

How easy is it to write programs in the language?

Factors effecting writability:

- Simplicity and orthogonality
- Support for abstraction
- Expressivity
- Fit for the domain and problem

Evaluation Criteria: Reliability

Factors:

- Type checking
- Exception handling
- Aliasing
- Readability and writability

Evaluation Criteria: Cost

Categories:

- –Programmer training
- –Software creation
- –Compilation
- –Execution
- –Compiler cost
- –Poor reliability
- –Maintenance

Evaluation Criteria: others

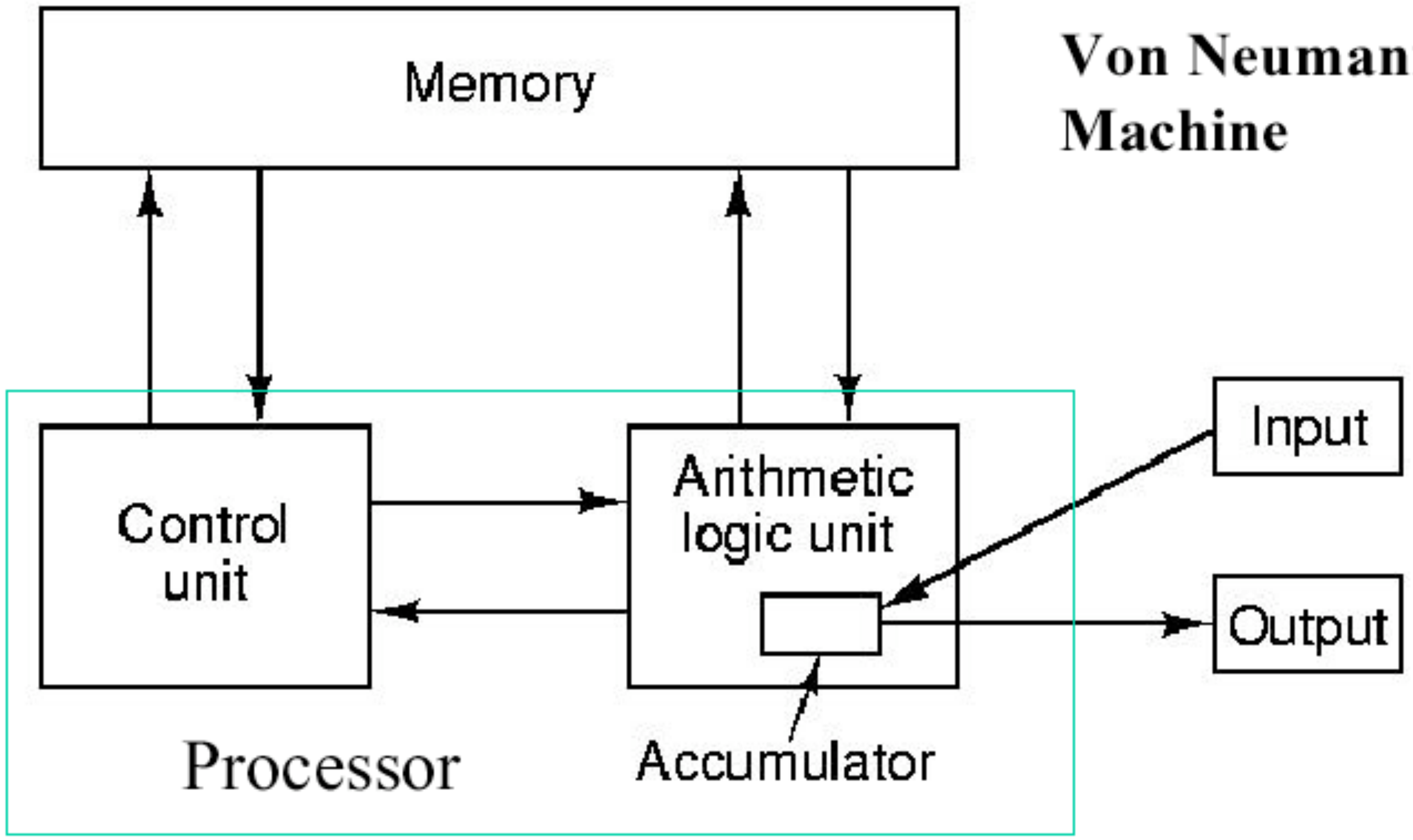
- Portability
- Generality
- Well-definedness
- Good fit for hardware (e.g., cell) or environment (e.g., Web)
- etc…

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)
…

Language Design Trade-offs

Reliability versus cost of execution

Ada, unlike C, checks all array indices to ensure proper range

Writability versus readability

(2 = 0 +.= T o.| T) / T <- iN

APL one-liner producing prime numbers from 1 to N

Flexibility versus safety

C, unlike Java, allows one to do arithmetic on pointers

Implementation methods

- **Direct execution by hardware**

  e.g., native machine language

- **Compilation to another language**

  e.g., C compiled to native machine language for Intel Pentium 4

- **Interpretation: direct execution by software**

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

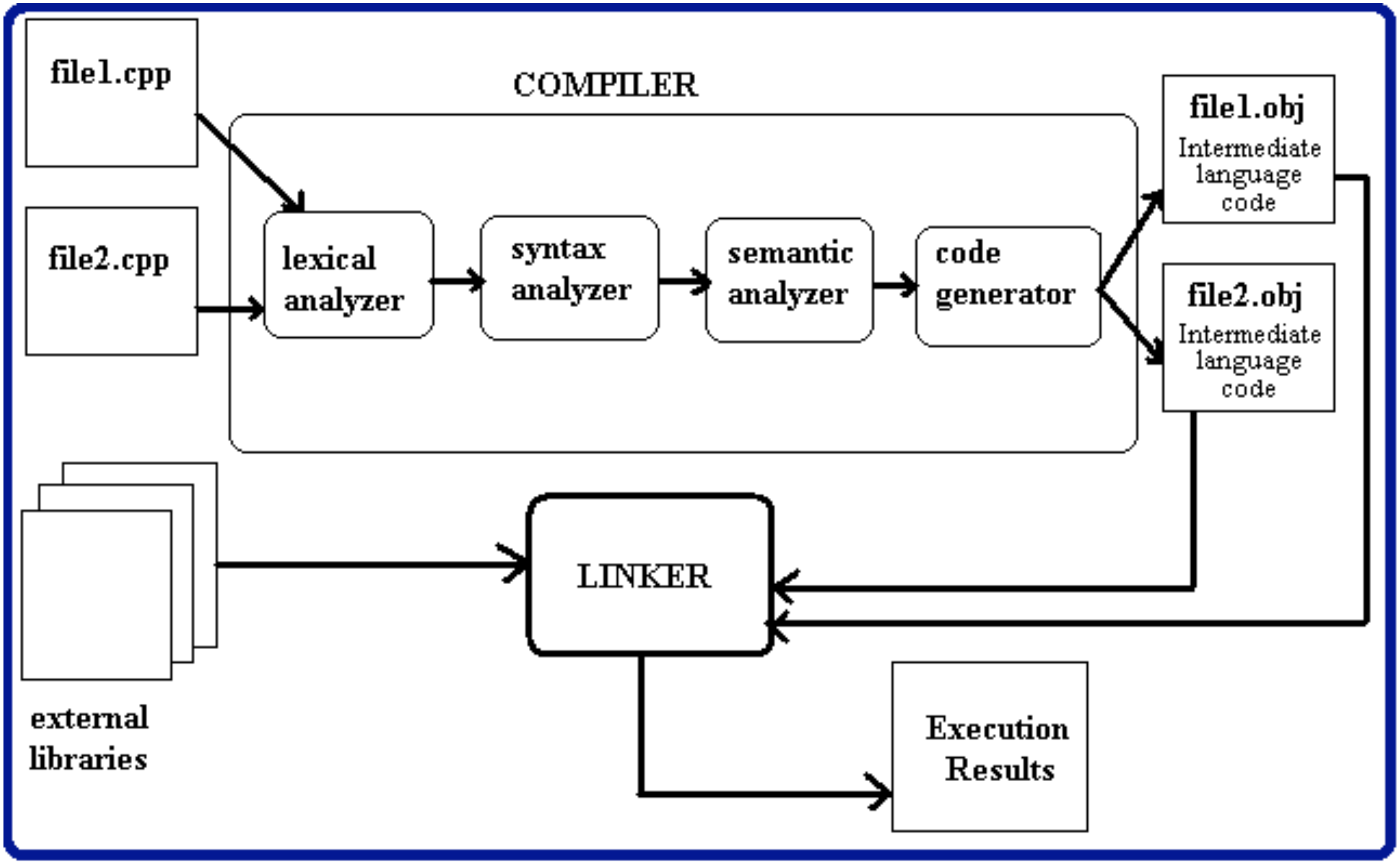
- **Hybrid: compilation then interpretation**

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

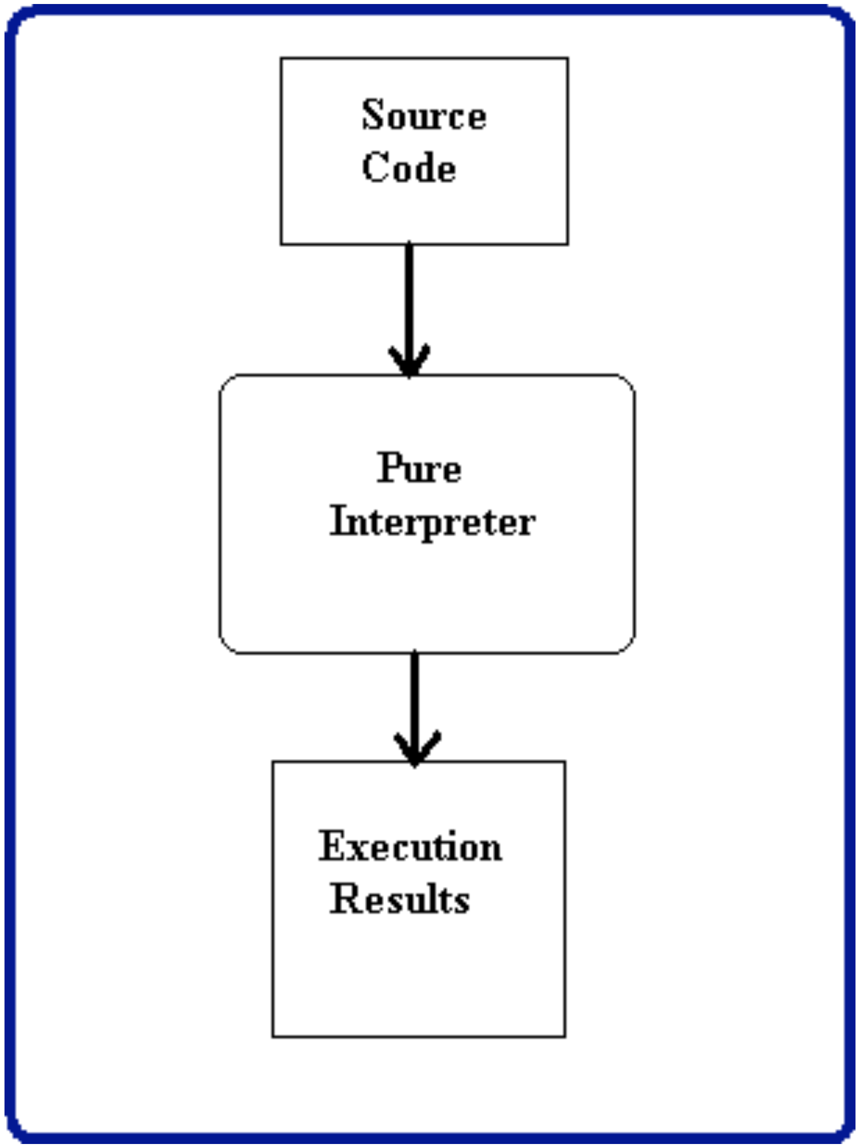
- **Just-in-time compilation**

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

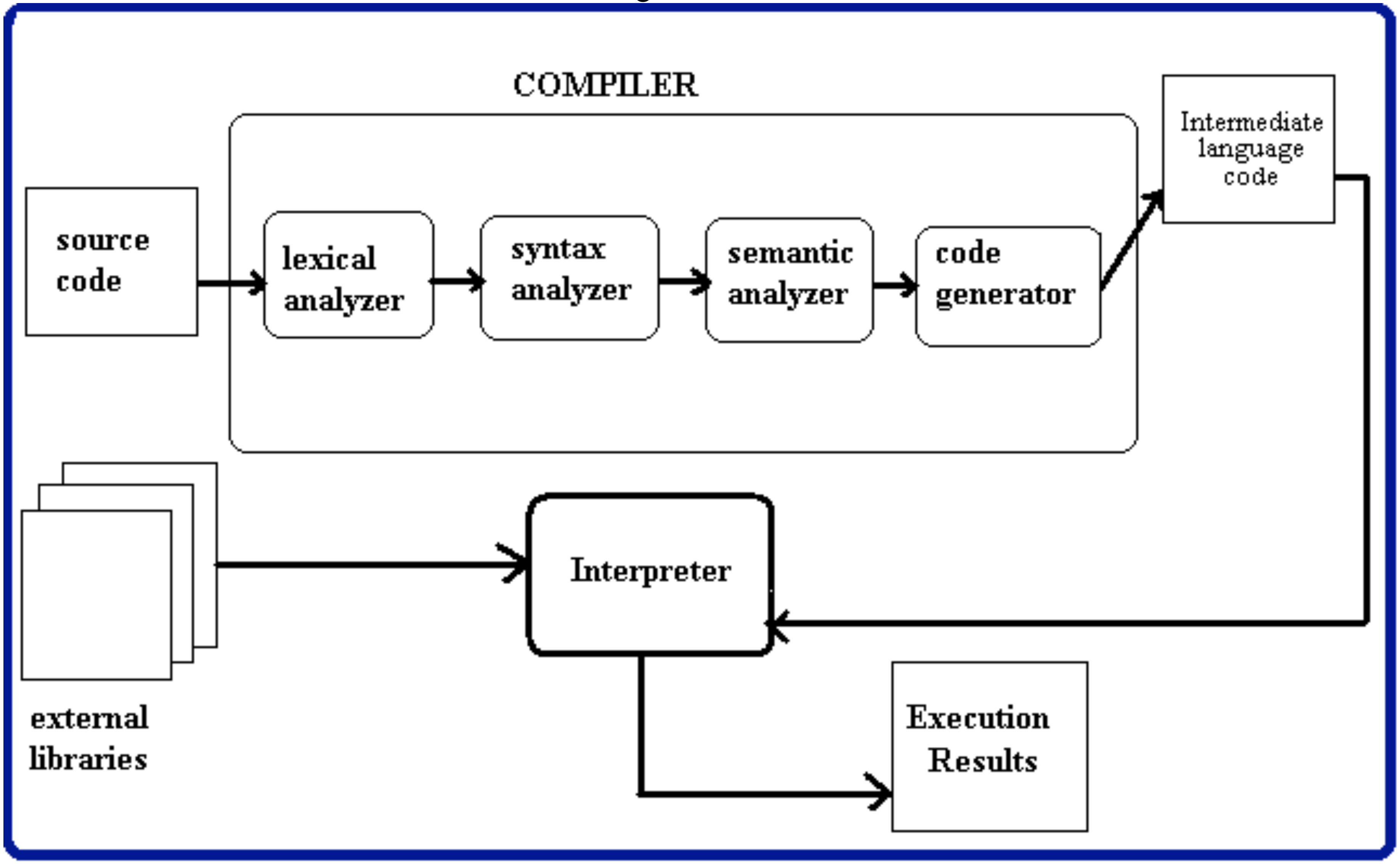
Compilation



Interpretation

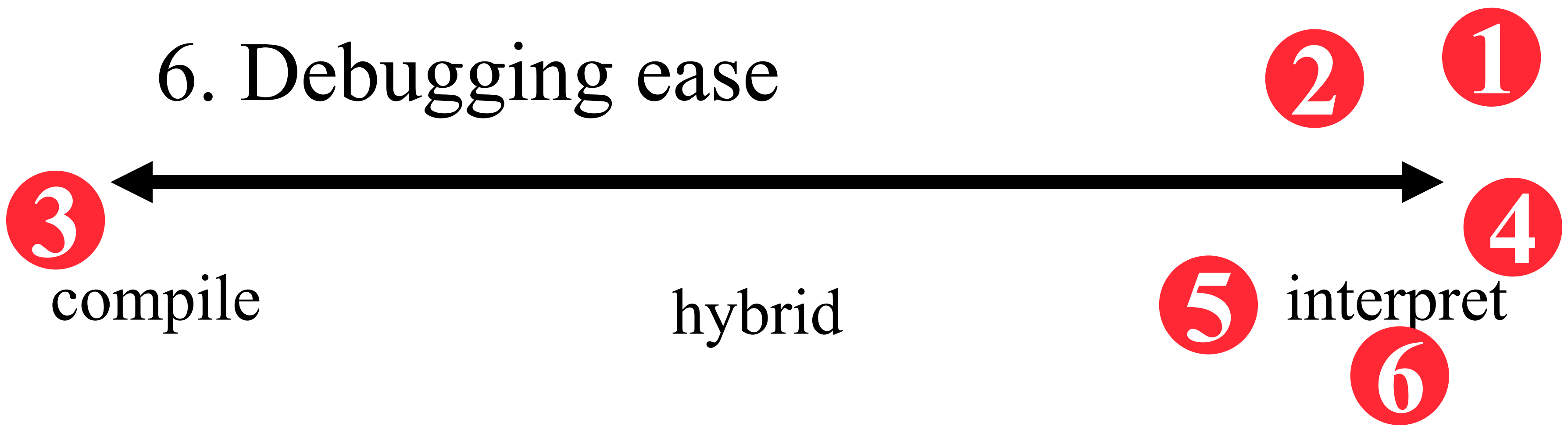


Hybrid


COMPILER
source code
lexical analyzer
syntax analyzer
semantic analyzer
code generator
Intermediate language code
external libraries
Interpreter
Execution Results

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