

Compilers and Programming Languages

An Introduction

PROS AND CONS

COMPILED		INTERPRETED	
ready to run	not cross-platform	cross-platform	interpreter required
often faster	inflexible	simpler to test	often slower
source code is private	extra step	easier to debug	source code is public

How are Languages Implemented?

* Translator could be classified as:

- Two major strategies:
 - (1) – Interpreters (older)
 - (2) – Compilers (newer)
 - (3) – A Combination
- without compilation step, going straight to the execution step.
- portability, etc. ①
- Good exe time, etc. ②
- ① + ② e.g. Java
- compilation step then to the execution step.

Good explanation about this division:
(Compiler - Interpreter - Hybrid)

<https://youtu.be/l1f45REi3k4>

- Interpreters run programs “as is”
 - Little or no preprocessing
- Compilers do extensive preprocessing

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend a natural language.

Tokenization Analysis

Lexical Analysis

→ same step also
done by the interpreter

- First step: recognize words.
 - Smallest unit above letters
 - This is a sentence.
- Lexical analyzer divides program text into “words” or “tokens”

If x == y then z = 1; else z = 2;

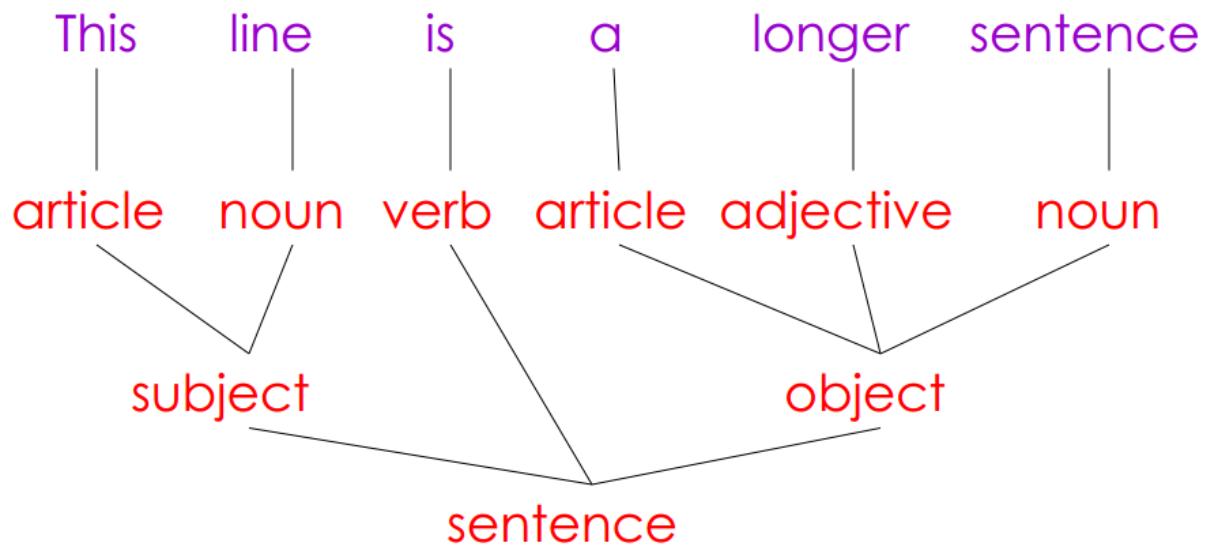
Key word identifier assignment integer

Syntax Analysis

Parsing

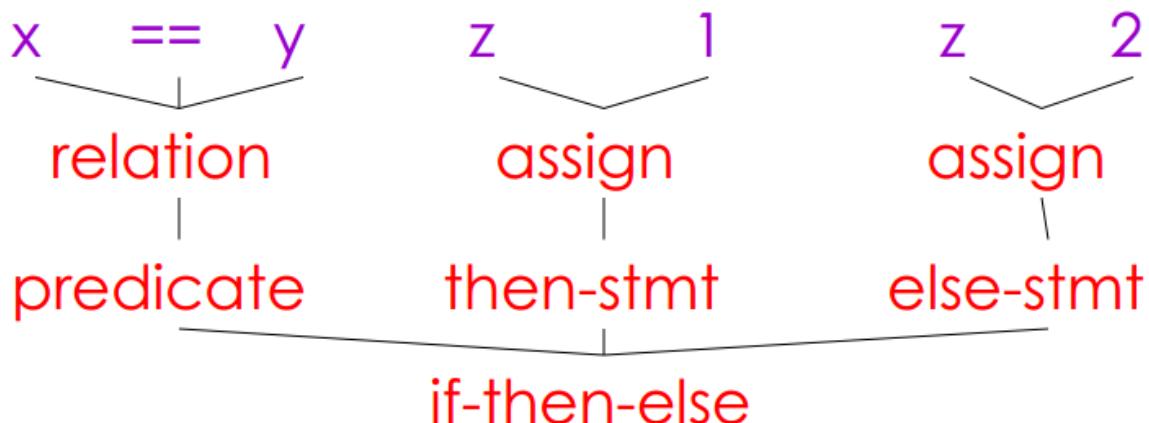
- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
 - The diagram is a tree

Diagramming Sentences



Diagramming Sentences

- Parsing program expressions is the same
- Consider:
 - If $x == y$ then $z = 1$; else $z = 2$;
- Diagrammed:



* the former steps are
for checking
the validation of the
code.

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
 - But meaning is too hard for compilers

Semantic issues:

①

because it needs
background knowledge and context

- Compilers perform limited analysis to catch inconsistencies.

Semantic Analysis in English

- Example:

Jack said Jerry left his assignment at home.

– What does “his” refer to? Jack or Jerry?

- Even worse:

Jack said Jack left his assignment at home?

– How many Jacks are there?

– Which one left the assignment?

Semantic Analysis in Programming

- Semantic Analysis in Programming

```
{
```

```
int Jack = 3;
```

```
{
```

```
int Jack = 4;
```

```
cout << Jack;
```

```
}
```

```
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example:

Jack left her homework at home.

②

 - A “type mismatch” between her and Jack; we know they are different people (Presumably Jack is male).

Optimization

- No strong counterpart in English, but akin to editing
 - You can express the same idea using fewer words
- Automatically modify programs so that they
 - Run faster *→ efficient*
 - Use less memory *→ memory requirements*
 - In general, conserve some resource

Optimization Example

- $X = Y * 0$ is the same as $X = 0$

Code Generation

- Produces assembly code (usually)
- A translation into another language
 - Analogous to human translation

Issues

- Compiling is almost this simple, but there are many pitfalls.
- Example: How are erroneous programs handled?
- Language design has big impact on compiler
 - Determines what is easy and hard to compile
 - many trade-offs in language design

 between language
design and compilation

Compilers Today

- The overall structure of almost every compiler adheres to our outline
- The proportions have changed since FORTRAN
 - Early: lexing, parsing most complex, expensive
 - Today: optimization dominates all other phases, lexing and parsing are cheap

Why So Many Languages?

- Application domains have distinctive and conflicting needs.
- E.g. For scientific applications you need
 - floating point representation
 - Arrays and operations on arrays
 - Parallelism
- For business Applications, you need
 - Persistence (you do not want to lose your data)
 - Report generation
 - Data analysis (i.e. you need to be able to ask questions about your data)
 - → SQL

- System Languages e.g.
 - Embedded systems
 - Operating systems
- For such applications, you need
 - Very low level control over resources
like working with Registers
 - Real time constraints, you need to be able to reason about time

Why are there new programming languages?

- Programmer trainer is the most dominant cost factor for a programming language
- → Widely used languages are slow to change (because any change required educating a large number of people)
- → Easy to start a new language (0 users = 0 training cost to begin with)
 - People will convert to the new language if productivity > training cost
- Languages are developed to fill a void as new technology appears e.g. mobile devices, internet, etc
- New languages look a lot like old languages (to reduce the training cost)

* Why were modular languages developed:

- To support huge applications.
- Divide and conquer approach for development.
- Data abstraction.

What is a good programming language?

- There is no universally accepted metric for language design.
- i.e.. people tend to disagree on what makes a good programming language
- 4 criteria

Language Evaluation Criteria

- ▶ **Readability:** maintenance is considered to be the major part of the software lifecycle, **and maintainability is directly related to how easily a program may be read.**
- ▶ **Writeability:** this is the measure of how easily a language may be used to create programs, and is closely related to readability.
Tools that make writing the program easier
more readable → more reliable
- ▶ **Reliability:** a program is reliable if it **performs to its specification under all conditions.**
more writable → more reliable
- ▶ **Cost:** the ultimate total cost.

Evaluation Criteria: Readability

- Overall simplicity
 - A manageable set of features and constructs *and constraints.*
 - Minimal feature multiplicity (small variety)
 - Minimal operator overloading → *e.g. operand like + for java it sets to addition and concatenation. we don't want so much operator overloading. overloading like using + for so many things. like push in stack etc.*
- Orthogonality
 - A relatively **small set of primitive constructs** can be combined in a relatively small number of ways to get the desired results.
 - The more orthogonal the design, the fewer exceptions.
 - Example In IBM assembly lang there are two instructions for addition
 - A Reg1, memory_cell
 - AR Reg1, Reg2
 - However in VAX's lang there is one that is more general
 - ADDL operand1, operand2
 - VAX's instruction for addition is more orthogonal than the instructions provided by IBM; hence, it is easier for the programmer to remember (and use) than the one provided by IBM.
 - This makes it easier to learn, read and write programs in a programming language.

→ Every possible combination is legal → fewer exceptions

(And whatever possible combination of primitives that we wish to use are legal & meaningful)

* Means also that a language feature is independent of where we use it in language
(For a programming language to be orthogonal, language constructs should not behave differently in different contexts).] examples:

Examples of Nonorthogonalities

- Other examples of nonorthogonalities include:
 - In Pascal functions can only return scalar values or pointers.
 - 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).

orthogonality in programming : means that operations change just one thing without affecting others. (operations are independent from each other)
(General Definition)

Evaluation Criteria: Readability

- Data types
 - Adequate predefined data types
- Syntax considerations
 - Identifier forms: flexible composition → older languages (like FORTAN) restrict the length of identifiers, which become less meaningful. Also not naming identifier by the name the keyword.
 - Special words and methods of forming compound statements
 - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
 - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
 - The ability to define and use complex structures like data structures or operations in ways that allow details to be ignored
 - ↓
not available for
public. like private
in java.
- Expressivity
 - A set of relatively convenient ways of specifying operations e.g. supporting for recursion.
 - ↓ Strength and number of operators and predefined functions

* And this doesn't conflict with the point of simplicity, since we mean here the predefined functions, and we meant there the constructs (like if, while...).

Evaluation Criteria: Reliability

- Type checking
 - Testing for type errors
- Exception handling
 - Intercept run-time errors and take corrective measures
- Aliasing – (bad in terms of reliability)
 - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
 - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability

java is very
good at it.

related to
programmer time

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system:
availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

related
to CPU
time

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
in opposite to C language, that choose cost of execution over reliability, because the CPU restriction and weakness in the time of establish it.
- 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
**but in general, if language is good with respect to readability
it's good with respect to writability and viceversa.*
- Writability (flexibility) vs. reliability
 - Example: C++ pointers are powerful and very flexible but are unreliable

Evaluation Criteria: Others

- Portability
 - The ease with which programs can be moved from one implementation to another
- Generality
 - The applicability to a wide range of applications
- Well-definedness
 - The completeness and precision of the language's official definition

Assessment characteristics

		Criteria	
Characteristic	Readability	Writeability	Reliability
Simplicity/orthogonality	•	•	•
Control structures	•	•	•
Data types and structures	•	•	•
Syntax design	•	•	•
Support for abstraction	✓	•	•
Expressivity	✓	•	•
Type checking			•
Exception handling			•
Restricted aliasing			•



Influences on Language Design

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

Influences on Language Design

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

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
 - Data and programs stored in memory
 - Memory is separate from CPU
 - Instructions and data are piped from memory to CPU
 - Basis for imperative languages

They appear as a result of von Neumann architecture

- Variables model memory cells

- Assignment statements model piping

- Iteration is efficient

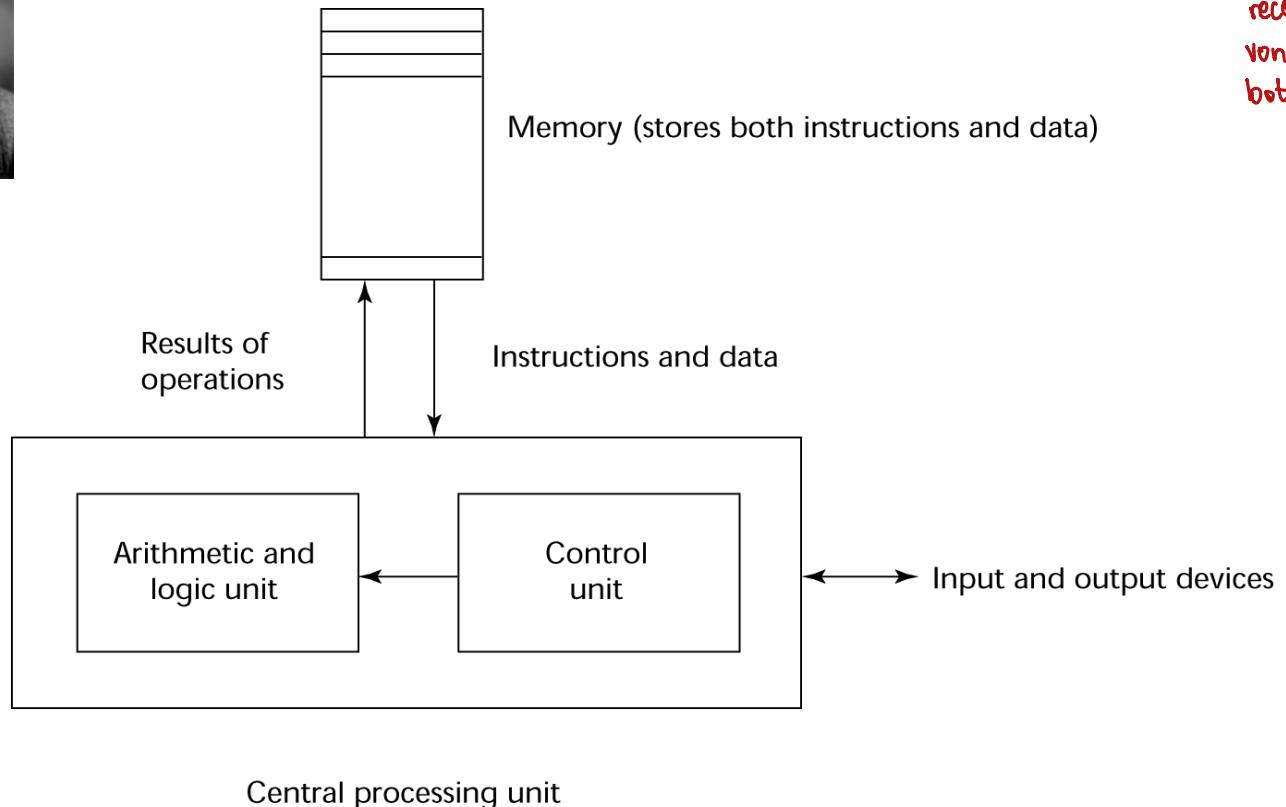
* Selection and branching statements (like if, switch).

done by changing the program counter (PC).

The von Neumann Architecture



*check the recording for von Neumann bottleneck.



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 ← done by the control unit
 in CPU.

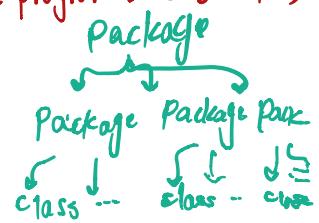
 execute the instruction

end repeat

store the result in the
memory (done when needed)

Programming Methodologies Influences

- 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 (*break the program to subtasks*)
- Late 1970s: Process-oriented to data-oriented
 - data abstraction
- Middle 1980s: Object-oriented programming
 - Data abstraction + inheritance + polymorphism



Language Categories

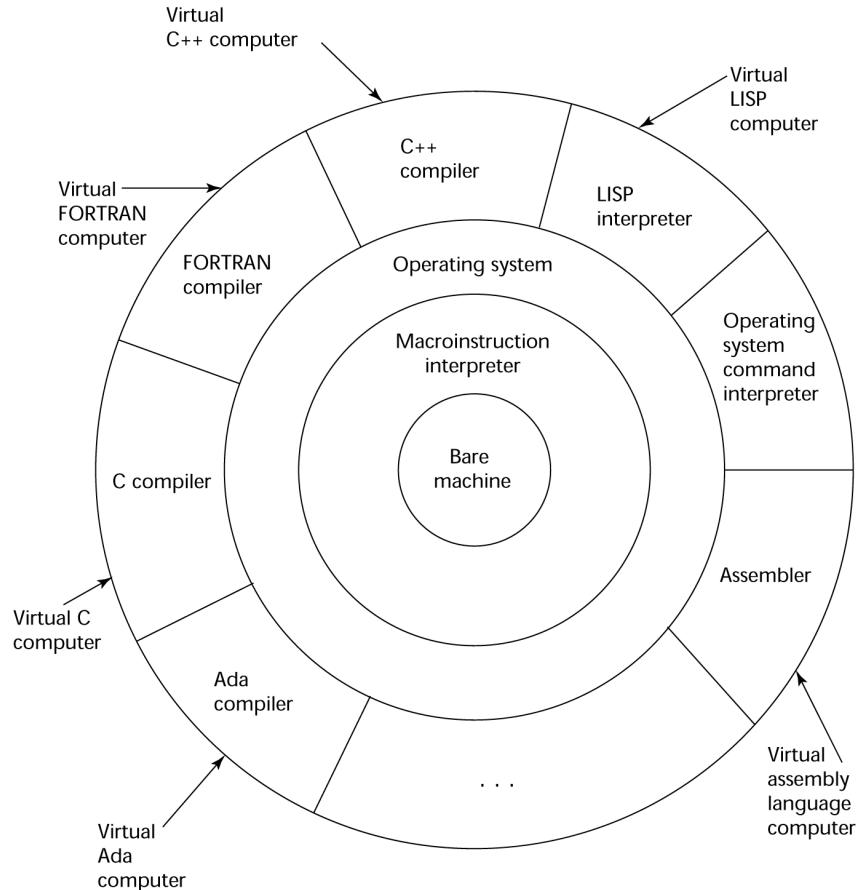
- Imperative → uses statements that change a program's (machine) state. Consists of commands for the computer to perform, and focuses on describing how a program works.
 - Central features are variables, assignment statements, and iteration
 - Include languages that support object-oriented programming
 - Include scripting languages
 - Include the visual languages
 - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional → in which the desired result is declared as the value of a series of function applications.
 - Main means of making computations is by applying functions to given parameters
 - Examples: LISP, Scheme
- Logic → in which the desired result is declared as the answer of conditions (rules).
 - Rule-based (rules are specified in no particular order)
 - Example: Prolog
- Markup/programming hybrid
 - Markup languages extended to support some programming
 - Examples: JSTL, XSLT (a language for transforming XML documents into other XML documents, or other objects such as HTML for web pages, plain text or into XSL Formatting Objects which can then be converted to PDF, PostScript and PNG. [2])

Implementation Methods

- Compilation
 - Programs are translated into machine language
- Pure Interpretation
 - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
 - A compromise between compilers and pure interpreters

Layered View of Computer

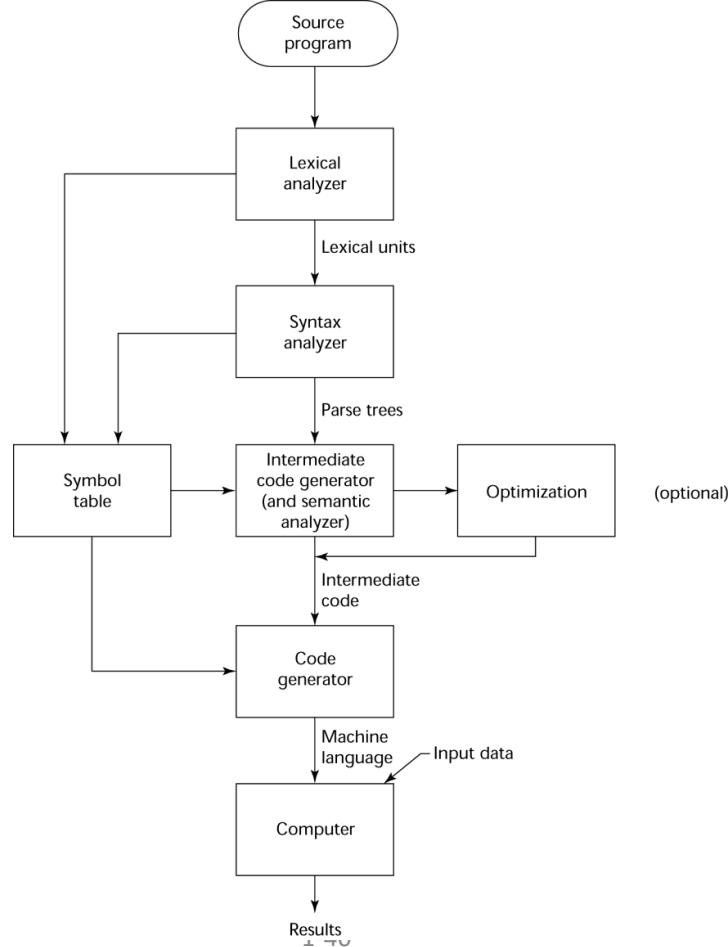
The operating system and language implementation are layered over machine interface of a computer



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



Phases of compiler:

https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm

Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

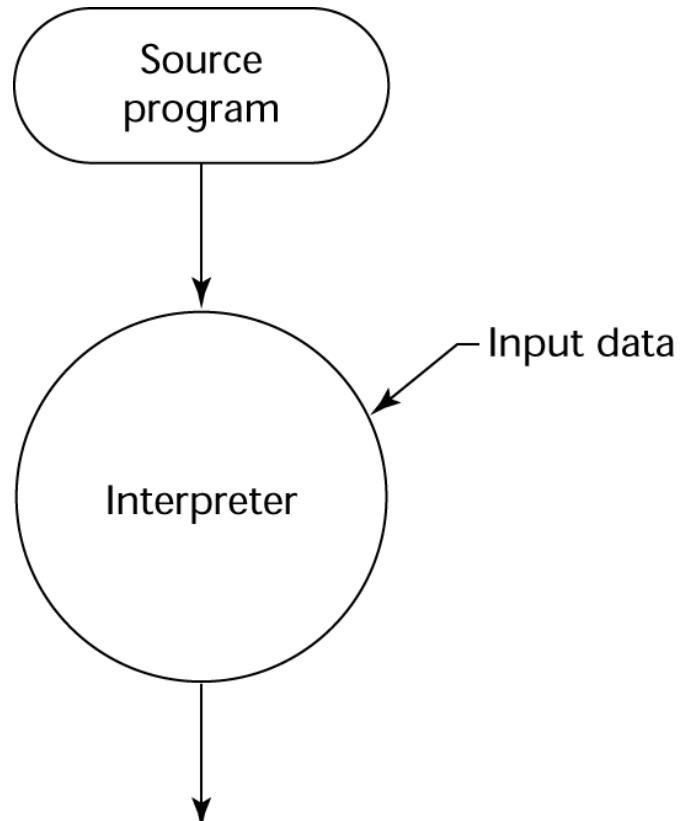
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program **instructions often can be executed much faster than the speed of the connection;** the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the **primary limiting factor** in the speed of computers

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

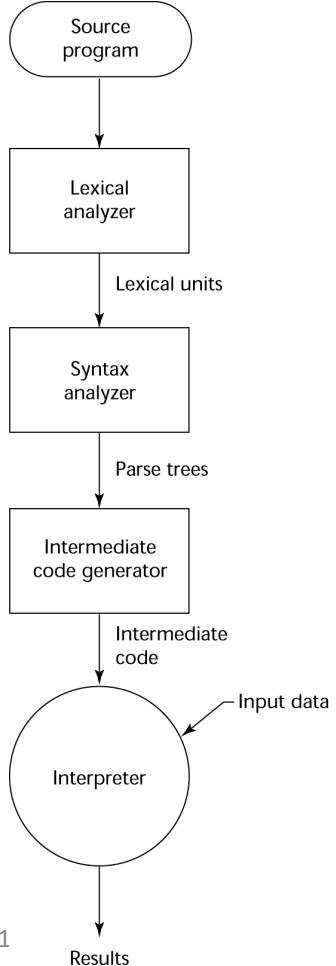


Compiler and Interpreter:
<https://youtu.be/e4ax90XmUBc>

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



Just-in-Time Implementation Systems

- Initially translate programs to an **intermediate language**
- Then **compile the intermediate language of the subprograms** into machine code when they are called
- **Machine code version is kept for subsequent calls**
- JIT systems are widely used **for Java programs**
- **.NET languages** are implemented with a JIT system

Why Study Languages and Compilers ?

- 1. Increase capacity of expression**
- 2. Improve understanding of program behavior**
- 3. Increase ability to learn new languages**
- 4. Learn to build a large and reliable system**
- 5. See many basic CS concepts a**