CSP3341 – Programming Languages and Paradigms

Lecture 1 – Preliminaries

Overview and Textbook Chapters

- This week we cover:
  - Preliminary topics and concepts of programming languages
  - Overview of programming language domains and categories
  - Language evaluation criteria and design influences
  - Language implementation and programming environments

- This week covers the following textbook chapter(s):
  - Chapter 1 – Preliminaries

  - *Reading the chapter(s) is required*

Why Should We Study Programming Languages?

- Learning to program is one thing, studying programming languages is another
  - Similar to learning to drive compared to learning how cars work

"The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general.

Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned."

– (Sebesta, p21)

Why Should We Study Programming Languages?

- Studying programming languages has many advantages:
  - Increased ability to express ideas
  - Improved background for choosing appropriate languages
  - Increased ability to learn new languages
  - Better understanding of significance of implementation
  - Better use of languages that are already known
  - Overall advancement of computing

- An obvious parallel exists between programming languages and natural languages
  - Studying programming languages is like studying Latin
  - Understanding *how languages work* makes learning a new language much easier
  - Vocabulary (syntax) of a language is not the important part

Programming Domains

- Computers have been applied to many different areas, falling into a few core domains
  - In modern times, the boundaries of these domains is blurry

- **Scientific Applications**
  - Original purpose of digital computers (1940s)
  - Involves lots of floating point computations
  - High use of arrays and matrices
  - Fortran was first and most prominent in this domain

- **Business Applications**
  - Business applications emerged in 1950s and 1960s
  - Produce reports, use decimal numbers and characters
  - First majorly successful business language was COBOL

Programming Domains

- **Artificial Intelligence**
  - Symbols rather than numbers manipulated
  - Use of linked lists rather than arrays
  - High flexibility often needed – e.g. Ability to create and run code during execution
  - LISP, a functional language, and Prolog, a logical language, often used for AI programming

- **Systems Programming**
  - Used for operating systems and must provide low-level features to interface with various external devices
  - Needs to be efficient because it is used continuously
  - C (and C-related languages) is the major language in this area. C is low-level and efficient, but relies on programmers having the knowledge/skill to use it safely and effectively

Programming Domains

- **Web Software**
  - Eclectic collection of languages – some dedicated to markup (e.g., XHTML, XML), which is not programming
  - Some web-enabled general-purpose languages (e.g. Java)
  - Scripting languages (e.g., PHP, JavaScript) designed for the web to provide dynamic content and web-based applications
  - A rapidly emerging and evolving field

Language Evaluation Criteria

- There are four primary factors to take into account when evaluating the features of a programming language
  - **Readability**: the ease with which programs can be read and understood
  - **Writability**: the ease with which a language can be used to create programs
  - **Reliability**: conformance to specifications (i.e., does what it is meant to do) in all conditions
  - **Cost**: the ultimate total cost

Readability

- Before 1970s, emphasis was on writability of code
  - As concept of software life-cycle emerged, the maintenance of code saw a more prominent role
  - Maintenance relies on readability, and hence from the 1970s onwards readability has become an important piece of criteria
- Appropriate language selection has impact on readability
  - If a language is used in a domain it was not intended for, it is likely to result in confusing and convoluted code

Factors Relating to Readability

- Overall Simplicity of Language
  - Manageable set of features and constructs
  - Avoid offering multiple ways to do the same thing
  - Avoid overloading operators or keywords in unintuitive ways
- Orthogonality
  - A relatively small and simple set of primitive constructs can be combined in a relatively small number of ways
  - All combinations are legal and have sensible/intuitive results
  - Minimal exceptions to these guidelines
- Data Types
  - Languages that offer appropriate data types are readable, e.g. “true” (boolean) is more meaningful than “1” (int)

Factors Relating to Readability

- Syntax Considerations
  - Variable name rules – short name limits hinder readability, requiring a special character at the start of variables can make them easier to spot, improving readability
  - Control structures that end group statements with **}** are less readable than those which use **end if**, **end loop**, etc
  - Meaning of reserved and key words in a language should be intuitive and obvious
  - Reserved and key words that have different meanings in different contexts reduce readability, as does the ability to define variables with the same names as reserved/key words

Writability

- Most of the factors that affect readability also affect writability – writing a program involves reading it quite a bit

- Again, must consider the intended domain of the language

Factors Relating to Writability

- Simplicity and Orthogonality
  - Large complex language with inconsistent rules for combining their primitive elements are more difficult to code in
  - Inefficient code may result if a user does not know of certain features of the language, or may use them incorrectly
- Support for Abstraction
  - Abstraction is a core concept in programming languages, allowing users to define and use structures that allow the details of a process to be ignored
  - Allows for efficient and meaningful code to be written
- Expressivity
  - Convenient and meaningful ways of specifying operations
  - Appropriate set of pre-defined functions to make use of

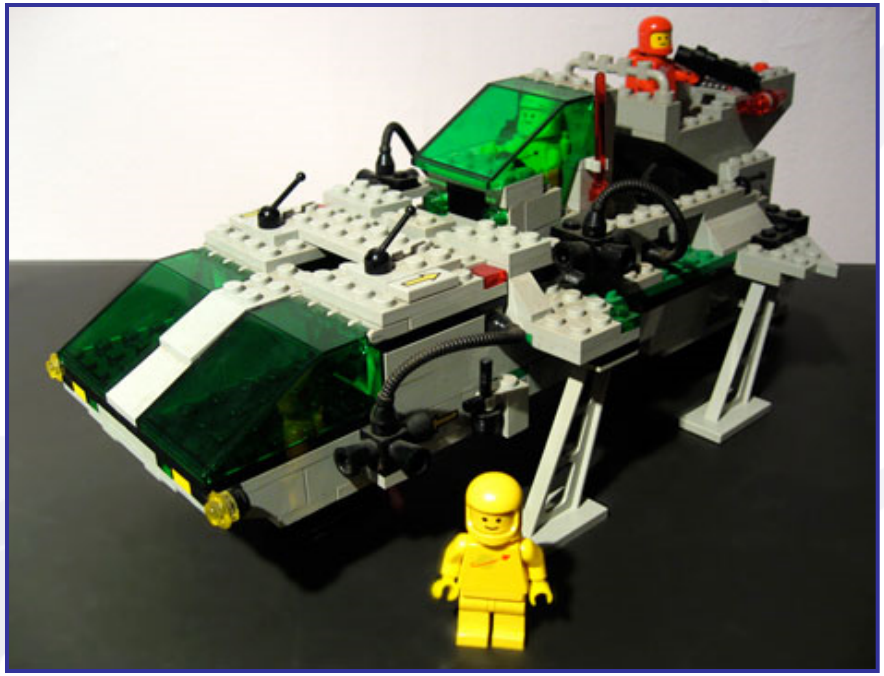
Readability and Writability

- Readability and writability are balancing acts – raising one can often lower the other, and both have extremes to avoid
- A language with massive amounts of pre-defined functions may be very easy to write, but can be difficult to read
- A language which is very simple may be highly readable, but too much simplicity can make it cumbersome to write in
- The goal is to strike an optimal balance between readability and writability – a language that is simple and predictable (readability), but flexible and expressive (writable)

Simplicity, Orthogonality, and Lego

- One way to think of simplicity and orthogonality in programming languages is by comparing it to Lego sets

- Old Lego sets were simple
  - Generic set of well-known pieces
  - Simple and consistent combination



- Some new Lego sets are silly
  - Lots of one-off and specific pieces
  - Limited connectivity and scope

Reliability

- A program is reliable if it performs to its specifications under all conditions
  - i.e. If it always does what it's meant to do
- Once again, a language's intended domain influences the reliability of a language
  - A program written in an inappropriate language/domain will be convoluted and inelegant, making it prone to errors
- Similarly, readability and writability influence reliability
  - A language which is hard to write in requires convoluted and inelegant solutions, making it prone to errors
  - A language which is not readable is harder to write in and harder to maintain, again making it prone to errors

Factors Relating to Reliability

- Type Checking
  - Languages which do not require data types to be declared and adhered to, or coerce variable types in expressions are less reliable than those which are strongly typed
  - Loosely typed languages are less able to detect type errors which could lead to erroneous program results or crashes
- Exception Handling
  - Languages which provide features allowing it to detect and handle run-time errors are more reliable
- Aliasing
  - Allowing more than one distinct name to refer to/access the same memory cell, e.g. Two pointers pointing at one variable
  - Reduces reliability and is a “dangerous” feature to use

Cost

- The ultimate total cost of a language is based on numerous factors, including readability, writability and reliability
- It is important to consider all aspects that contribute to cost, e.g. Time, effort, need for expertise. Such aspects may not be directly financial, but all have their associated cost
  - Highly readable languages are quicker and easier to maintain, minimising the cost of maintaining/updating code
  - Highly writable languages are quicker and easier to code in, minimising the cost of developing code
  - Highly reliable languages do not need to be maintained as often, minimising the cost of maintaining/updating code
    - The failure of a mission-critical program could be very costly

Factors Relating to Cost

- Other factors affecting the cost of a language include:
  - The cost (in time) of learning or being trained to use a language (influenced by readability and writability)
  - The cost (in time and resources) of compiling and executing code was important in early languages, but less so now
  - Optimisation during compilation is a trade-off between compile cost and execution cost – during development when compilation is frequent, optimisation is less cost effective
- Maintainability of code, which relies on all three of the other criteria, is arguably the most important cost factor

Language Evaluation Criteria Summary

- Cost is largely a product of the other three criteria
- Readability and writability can be seen as the two most important pieces of criteria
  - They both have a significant impact on reliability and cost
- The importance of certain criteria can vary depending on the domain and type of application at hand
  - Mission-critical and embedded programs must be reliable
  - Languages hoping for widespread use in industry and education should be readable and writable
- Language designers must strive for an optimal balance of all criteria, relative to the intended application/domain

Influences on Language Designs / Methodologies

- In 1950s and early 1960s, hardware was expensive and using it was time-consuming and costly
  - Languages focused on machine efficiency
- From the late 1960s, as hardware became less expensive and more powerful, person efficiency became the focus
  - More sophisticated languages and programs
  - Programmer costs increase with complexity of programming
  - Better control structures and type checking emerge
- In the late 1970s, data-oriented design emerged
  - Focus on abstract data types
- Early to mid 1980s, object-oriented design emerged
  - Improved efficiency and via inheritance and polymorphism

Language Categories

- There are several core categories of languages

- **Imperative** languages are the most common/widely used
  - Includes scripting languages (typically only distinct by their implementation method)
  - Includes most object-oriented languages (object-oriented programming differs from procedural programming, but the languages that support it still tend to be imperative)

  - Involves sequences of statements that change a program state (i.e. “do stuff”) – a *procedural* approach, defining *how* to achieve the desired results
  - Heavy use of subprograms/functions to abstract processes
  - Includes FORTRAN, BASIC, Ada, Java, C, Python, PHP...

Language Categories

- **Functional** languages are related to lambda calculus
  - Design based on evaluation of mathematical functions
  - Fundamentally different from imperative languages – computation via evaluation of mathematical functions, no need for variables, not based on changing state of program
  - Declarative rather than imperative – statements describe *what* to accomplish, rather than how to accomplish it
  - Languages include LISP, APL and Scheme
  - Applications are primarily in artificial intelligence

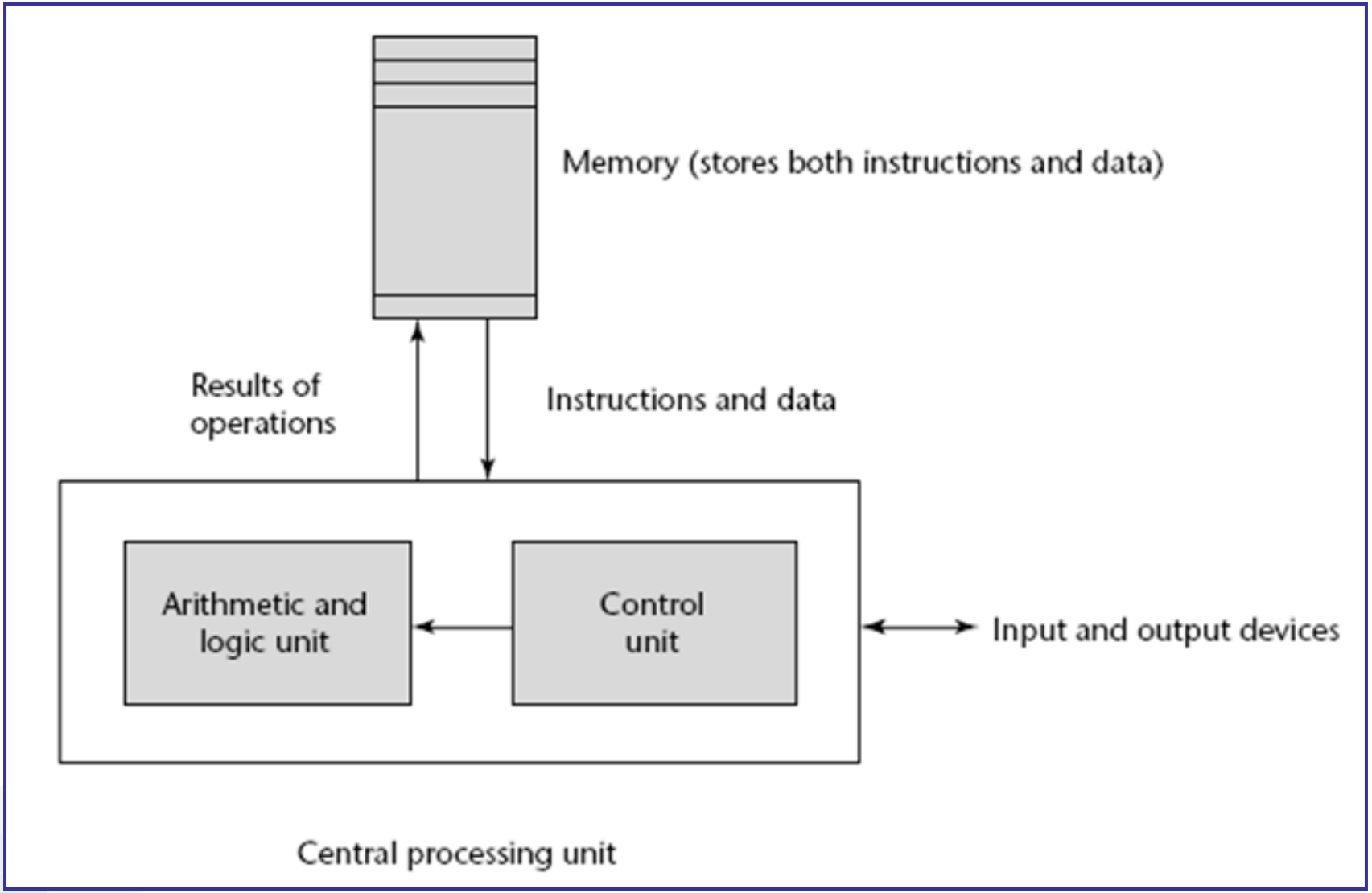
Language Categories

- **Logic** languages are related to predicate calculus
  - Design based on evaluation of mathematical logic
  - Fundamentally different from imperative languages – Express programs in a form of symbolic logic
  - Also declarative, describing *what* not *how*
  - Languages include Prolog
  - Applications are primarily in expert systems and natural language processing
- Functional and logic languages have simpler syntax and semantics, but are inefficient to execute. Both are niche

The Von Neumann Computer Architecture

- The prevalent computer architecture since their widespread inception is the von Neumann (“von Noyman”) architecture
  - This has had a profound effect on the design of languages, which were made to execute efficiently on this architecture
- Fundamentals of the architecture involve:
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - CPU executes instructions and returns results to memory
- Led to the fundamental concepts of imperative languages – from variables and assignment statements to iteration
  - Functional and logic languages are not based on this architecture, and hence their execution is relatively inefficient

The Von Neumann Computer Architecture



– Sebesta, 2009, p38

The Fetch-Execute Cycle

- The process of executing machine code on a computer of the von Neumann architecture is the fetch-execute cycle

- It is essentially an endless loop involving:
  - *Fetch the next instruction* from memory
  - *Decode the instruction* to determine what action it specifies
  - *Fetch data from memory* if the instruction requires it
  - *Execute the instruction*
  - *Return results to memory*

- Control of the cycle/instruction queue transfers between the operating system and applications as required

The Von Neumann Bottleneck

- Speed of the connection between a computer's memory and its CPU usually determines the speed of the computer
  - This is because instructions can typically be executed faster than they can be moved to the processor for execution
  - The CPU is often idle, waiting for the next instruction to arrive
  - This is known as the von Neumann bottleneck, and is the primary limiting factor of the von Neumann architecture
  - The von Neumann bottleneck is a core motivator for research into parallel computing, where multiple CPUs are available

Language Categories and Computer Architecture

- Early programming languages and all of the widespread/ commonly used languages are all imperative
  - Imperative languages are closely tied to the von Neumann architecture, designed for efficiency using that architecture
- Functional and logic languages are not based on this
  - Many feel that functional and logic languages are technically more sophisticated in terms of syntax and semantics, etc
  - Yet such languages have never seen more than niche usage
- Despite supposed technical superiority, functional and logic programming are unlikely to become more than niches
  - Even if functional and logic languages were as efficient as imperative languages, people are accustomed to imperative

Language Implementation Methods

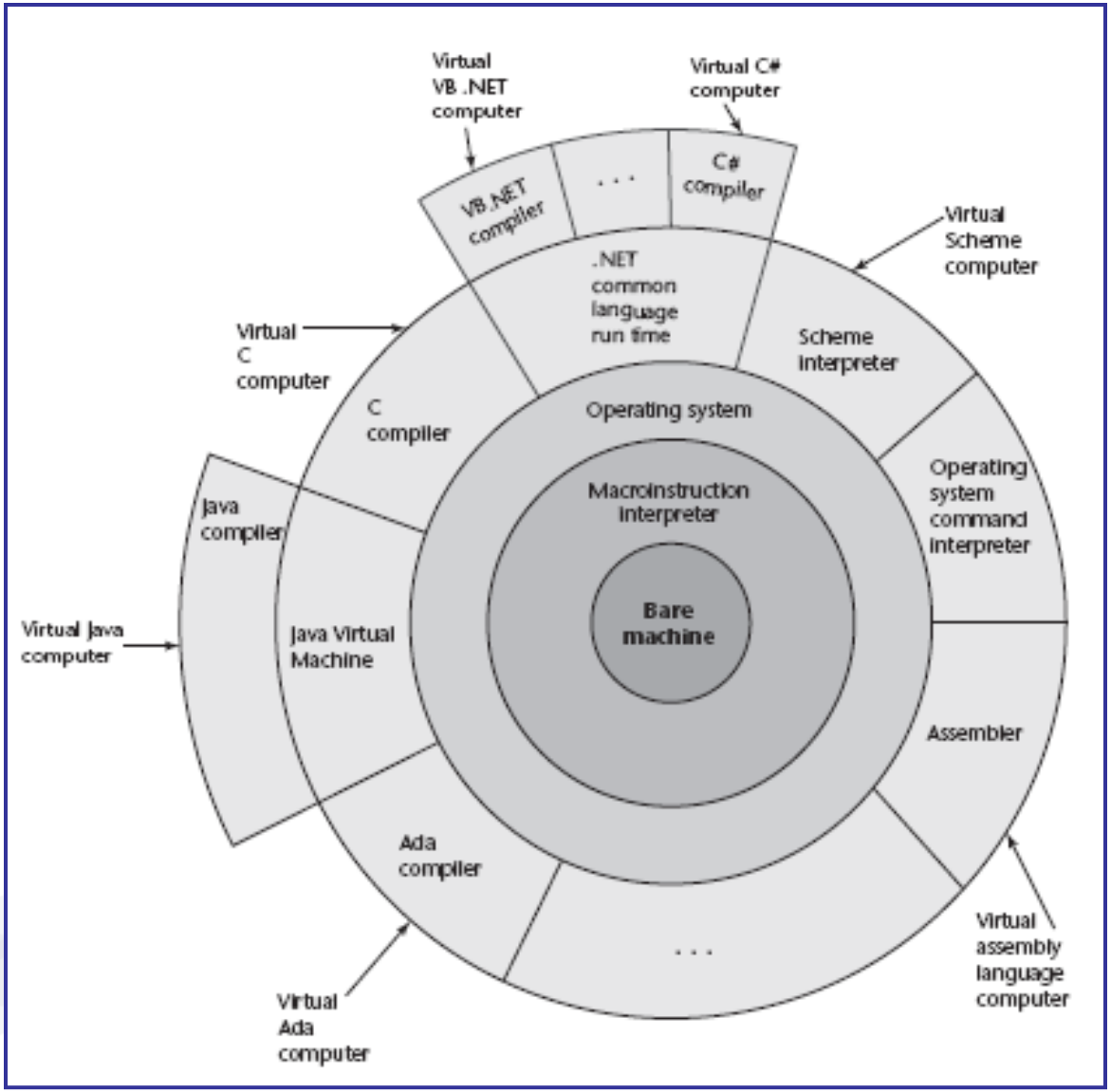
- **Compilation**
  - Code is translated into machine language by compiler
- **Interpretation**
  - Code is interpreted by a program known as an interpreter
- **Hybrid Implementation**
  - A compromise between compilers and interpreters

Machine Code and High-Level Language Interfaces

- CPU only understands machine code – a very low-level language that provides the most basic primitive operations

- System software is needed to provide an interface to this, translating higher level languages to machine code
  - The OS provides interfaces with input and output devices, resource management, storage, etc
  - High-level languages need these, hence their interfaces interact with the OS, rather than directly with the CPU
  - These interfaces for high-level languages are layered over the OS and can be thought of as **virtual computers**
  - This is a form of abstraction – the implementation/execution details of lower level languages are hidden from higher levels

Layered Virtual Computers



– Sebesta, 2009, p44

Compilation

- Compilation is a language implementation method whereby the source code is translated into machine code which can be executed directly on the computer

- Compilation process has several phases:
  - Lexical analysis: converts 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

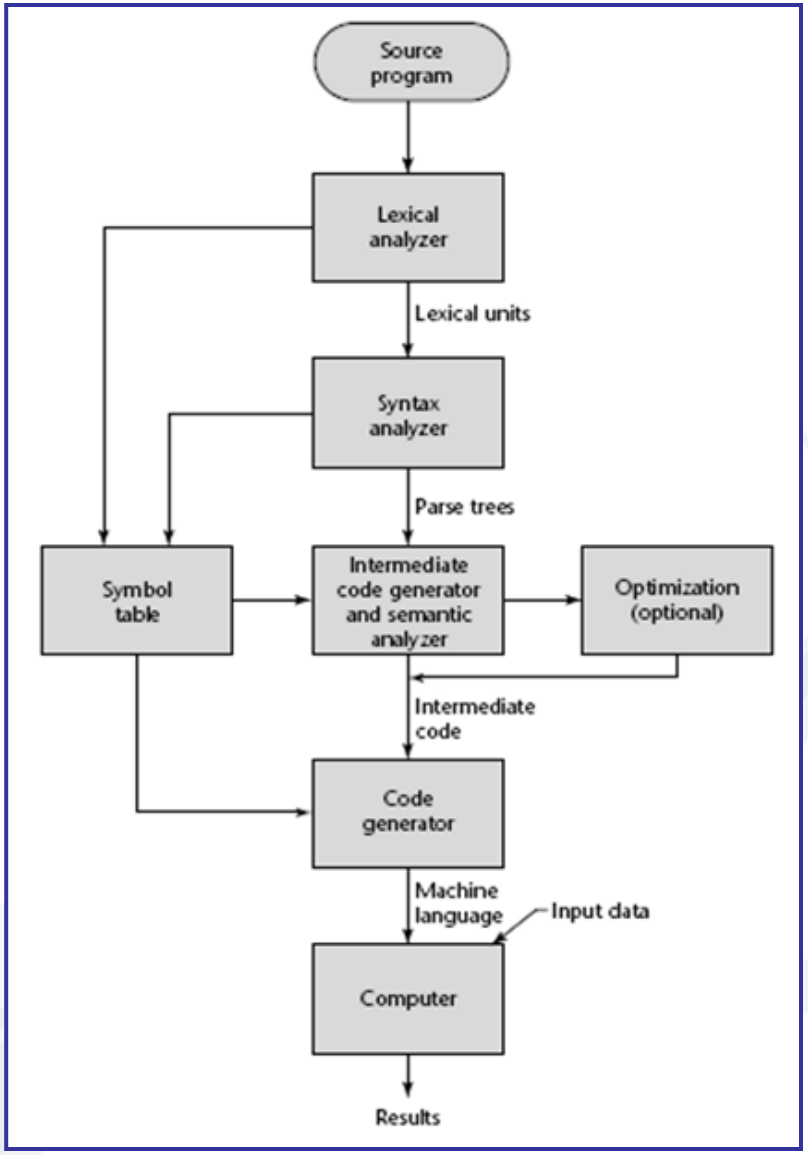
- The translation process is slow, but results in very fast program execution of compiled programs
  - Compilation can include optimisation, which results in even faster and/or smaller compiled programs

Interpretation

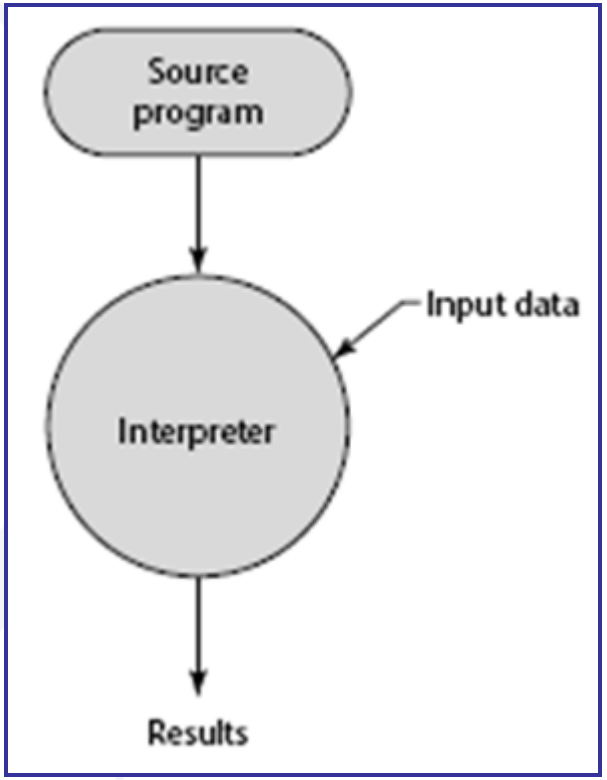
- Interpretation is at the opposite end of the scale from compilation. The source code of a program is interpreted by an interpreter without any translation
- Interpreter is a software simulation of a processor performing a fetch-execute cycle for the high-level language
- No translation means run-time debugging is much easier
- Execution is much slower than a compiled program due to complexity of decoding high-level statements
- Rarely used in high-level languages since the 1980s, but has made a resurgence in web-based scripting languages

Compilation and Interpretation Processes

- Compilation process



- Interpretation process

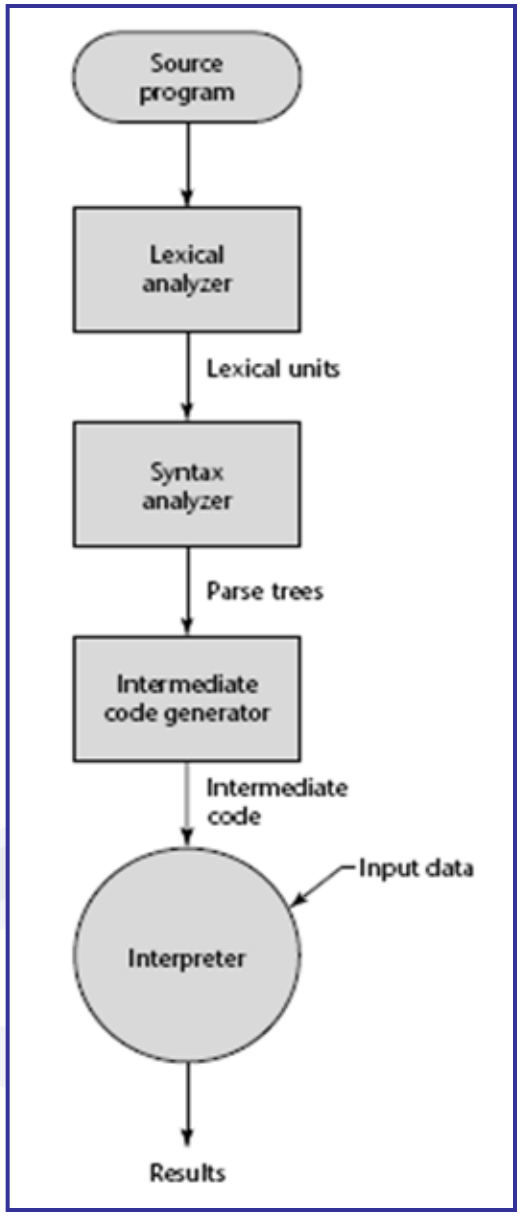


– Sebesta, 2009, p45, p48

Hybrid Implementation

- Hybrid implementation compromise between compilation and interpretation. High-level language is translated to intermediate language, which can be interpreted easier
- Faster than pure interpretation, as high-level statements only need to be decoded once, into intermediate language
- Initial Java implementations were hybrids:
  - Source code compiled into intermediate “byte code”
  - Byte code could be interpreted on any machine that has the Java Virtual Machine installed, containing the interpreter
  - This results in high portability of Java programs

Hybrid Implementation Process



– Sebesta, 2009, p49

Just-In-Time Implementation

- JIT implementations initially translate source code into an intermediate language, and then compile the subprograms of it into machine code when they are called
  - Compiled versions kept for subsequent calls to save time
  - High-level statements only need to be decoded once
  - Individual subprograms can be recompiled if needed
- The JIT process is essentially compilation, with a pause and some more flexibility
- Widely used in modern Java and .NET implementations

Programming Environments

- The collection of tools used in software development is known as the programming environment
- Some languages include development environment tools, and third-party vendors also create such tools
- Not part of the language itself, but still an important factor
  - A sophisticated and usable programming environment makes writing, reading and maintaining code much easier
- Specific examples include Microsoft Visual Studio .NET and NetBeans, but generic tools such as programming-oriented text editors also exist

Summary

- Justification for the study of programming languages
- Programming domains
- Language evaluation criteria
  - Readability, writability, reliability and cost
- Influences upon language design
- Language categories
  - Imperative, functional, logic
- The von Neumann computer architecture
  - Fetch-execute cycle and the von Neumann bottleneck
- Language implementation methods
  - Compilation, interpretation, hybrid implementation
- Programming environments