1. PRELIMINARIES

Reasons for Studying Concepts of Programming Languages

- Increased capacity to express ideas

  Language constructs can often be simulated in languages that do not support those constructs directly.

- Improved background for choosing appropriate languages

- Increased ability to learn new languages

- Better understanding of the significance of implementation

  Helps develop the ability to use a language as it was designed to be used
  Helps in fixing bugs
  Helps in understanding relative efficiency of alternative constructs

- Better use of languages that are already known

- Overall advancement of computing

  If those who choose languages were well informed, perhaps better languages would eventually squeeze out poorer ones.

Programming Domains

- **Scientific applications**

  Requirements: Simple data structures (arrays); large numbers of floating-point calculations; efficiency.
  Languages: Fortran, ALGOL 60.

- **Business applications**

  Requirements: Report generation; character data; decimal arithmetic.
  Languages: COBOL.

- **Artificial intelligence (AI)**

  Requirements: Symbol manipulation; linked lists; ability to create and execute code at run time.
  Languages: Lisp, Scheme, Prolog.

- **Web software**

  Languages: Java, JavaScript, PHP.

Language Evaluation Criteria

- **Readability**

  The development of the software life cycle concept in the 1970s caused a greater emphasis to be placed on maintenance, which is greatly affected by readability.

  Readability must be considered in the context of the problem domain.

- **Writability**

  Many of the same characteristics that influence readability also affect writability.

  Writability must also be considered in the context of the problem domain.

- **Reliability**

  A program is reliable if it performs to its specifications under all conditions.

- **Cost**

Readability

- **Overall simplicity**

  A simple language has a relatively small number of basic constructs.

  Having multiple ways to accomplish the same effect (**feature multiplicity**) can complicate a language. In Java, there are four ways to increment a variable:

  ```
  count = count + 1
  count += 1
  count++
  ++count
  ```

  Programmer-defined **operator overloading** can lead to complexity.

  Too much simplicity can be a problem; a language may lack adequate control structures and data structures.

- **Orthogonality**

  Orthogonality means that a small set of primitive constructs can be combined in a small number of ways—with no restrictions—to build the language's control structures and data structures.

  Orthogonality is closely related to simplicity. A lack of orthogonality shows up in the form of exceptions to the rules of a language.

  C has numerous exceptions to its normal rules. For example, a function can return a structure (record), but not an array.

  Too much orthogonality can cause problems. ALGOL 68 is perhaps the orthogonal language ever designed; unfortunately, it allows many unusual and confusing combinations of features.

Readability (Continued)

- **Data types**

  C89 lacks a Boolean type, forcing the programmer to use integers to represent Boolean values. The meaning of an assignment such as `timeOut = 1` is not clear.

- **Syntax design**

  Readability is affected by a language's choice of special words.

  Many languages have no special marker for the end of a control structure other than the word `end` or a right brace. Ada, on the other hand, uses `end if` for the end of an `if` statement and `end loop` for the end of a loop.

  Some languages (including Fortran) allow special words to be used as identifiers, which can be confusing.

  Semantics (meaning) should follow directly from syntax (form).

Writability

- **Simplicity and orthogonality**

  A large number of features can lead to misuse of some features and disuse of others that are superior. Also, it is possible to use an unknown feature accidentally.

  Orthogonality helps writability, unless features are too orthogonal, in which case errors may go undetected.

- **Expressivity**

  A language is expressive if it provides features that make it easier to perform common tasks.

  APL provides powerful array operators. C++ provides increment and decrement operators. Many languages provide a `for` statement, which is more convenient than a `while` statement for writing counting loops.

Reliability

- **Type checking**

  Type checking means to check for type errors, either during compilation or during program execution.

  Run-time type checking is expensive, so compile-time type checking is preferred. Also, compile-time type checking catches errors earlier, when they are easier (and cheaper) to fix.

  Java requires extensive type checking at compile time. Compilers for the original C language, on the other hand, did not perform type checking on function parameters, either at compile time or at run time.

- **Exception handling**

  Exception handling allows a program to intercept run-time errors and take corrective action.

  Ada, C++, Java, and C# provide exception handling, but many older languages, including C, do not.

- **Aliasing**

  Aliasing occurs when there are two or more names for the same memory location. Aliasing can lead to difficulties in understanding and debugging a program.

  Most languages allow some kind of aliasing. In C, the use of unions and pointers can lead to aliasing.

- **Readability and writability**

  Programs written in a language that is not readable can be difficult to debug and modify.

  If a language lacks writability, programs written in it may be unnatural and more likely to contain bugs.

Cost

- The cost of using a programming language is affected by many factors.

- **Cost of training programmers**

  A simple and orthogonal language is usually easier to learn.

- **Cost of writing programs**

  The cost of writing programs depends on the writability of the language, which depends on how well the language suits the application.

  A good programming environment can help reduce the cost of writing programs (and the cost of training programmers).

- **Cost of compiling programs**

  A language that compiles too slowly may be too expensive to use. Some early Ada compilers were exceedingly slow.

- **Cost of executing programs**

  Programs written in a language that requires many run-time type checks will pay a performance penalty.

  Some languages are interpreted rather than compiled, which can have a big impact on execution time.

  Programs can often be optimized for either time or space. **Optimization** generally increases the time required for compilation, however.

- **Cost of language implementation system**

  For some languages, such as Java, good implementations are available at no cost.

  A language that requires an expensive implementation or expensive hardware will probably never be widely used.

Cost (Continued)

- **Cost of poor reliability**

  Programs that fail can cause huge costs in lost business, lawsuits, injuries, or even deaths.

- **Cost of maintaining programs**

  Most programs are modified over time to correct bugs, adjust to changes in specification, and add enhancements.

  For large systems with long lifetimes, the cost of maintenance can be two to four times as much as the cost of development.

  The cost of maintenance depends heavily on language readability.

- The most important contributors to language cost are program development, maintenance, and reliability. All three are functions of language readability and writability.

- Other criteria for evaluating programming languages:

  Portability (ease of moving programs from one implementation to another)
  Generality (suitability for a wide range of applications)
  Well-definedness (completeness and precision of the language's definition)

- Portability and well-definedness are influenced by the existence of a standard for the language.

- Standardization is a time-consuming and difficult process.

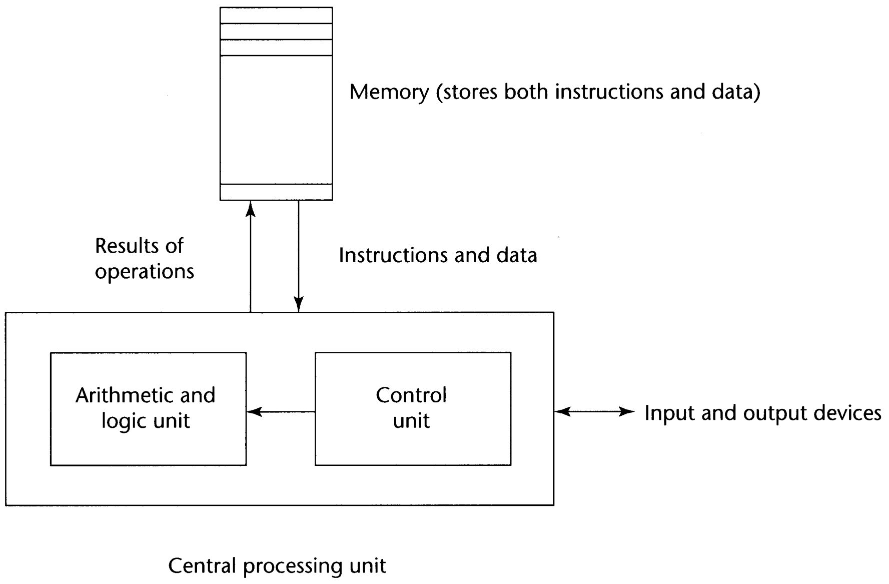
Influences on Language Design

- Programming languages have been strongly influenced by computer architecture and by programming design methodologies.

- **Computer architecture**

  Most popular languages of the past 60 years have been designed around the **von Neumann architecture**. These languages are called **imperative** languages.

  In the von Neumann architecture, instructions and data are stored in memory. Instructions are fetched from memory and executed by a central processing unit (CPU).



In an imperative language, variables represent memory cells. Assignment statements model the movement of data from memory to the CPU and back again.

Influences on Language Design (Continued)

- **Programming design methodologies**

  As programs became larger, methodologies began to emerge in the 1960s and 1970s to deal with the growing complexity of software:

  Top-down design and stepwise refinement
  Data abstraction
  Object-oriented design

  Each methodology had an effect on language design.

  Smalltalk helped popularize object-oriented programming. Most imperative languages now include support for object-oriented programming.

  More recently, growing interest in concurrency has had an influence on language design. Java and C# support concurrent programming.

Language Categories

- Programming languages are often put into four categories:

  Imperative
  Functional
  Logic
  Object-oriented

- Functional languages provide only the ability to define functions and call them. A pure functional language has no variables and no assignment statements.

- In a logic programming language, a program consists of a series of facts and rules.

- Object-oriented languages are often extensions of imperative languages.

- Markup languages, such as HTML and XML are not programming languages. However, programming capabilities can be added to these languages via JSTL and XSLT.

Language Design Trade-Offs

- Language evaluation criteria are often self-contradictory. A language designer must make trade-offs to achieve a satisfactory design.

- Conflict: Reliability versus cost of execution.

  Java performs extensive run-time checks, including checking whether array subscripts are within range. C omits such checks.

- Conflict: Writability versus readability.

  APL provides a powerful set of array operators, making it an expressive language and highly writable language (at least for programs that perform many array operations), but not a readable one. APL programs rely on unusual operators, often combined into complex expressions.

- Conflict: Writability versus reliability.

  Pointers in C++ are highly flexible, which makes it easier to write programs. On the other hand, errors in pointer usage are common and can lead to severe problems.

Implementation Methods

- Most computers are capable of understanding only a primitive machine language. As a result, executing programs written a high-level language requires a language implementation system.

- Language implementation systems, in turn, rely on the services of the computer's operating system, creating a set of layers.

- In the 1950s, language implementation systems were very complex. Research done in the 1960s and later has made them easier to write. Using special tools, it is now possible to automatically generate large portions of a language implementation system.

Compilation

- One implementation method for a programming language is compilation, in which a program written in a **source language** is translated to machine language by a **compiler**.

- Compilation has the advantage of fast program execution.

- Many common languages, including C, C++, and COBOL, are compiled.

- Phases of a compiler:

  Lexical analyzer—Gathers characters into lexical units (lexemes) and discards comments.

  Syntax analyzer (*parser*)—Checks the program for syntax errors. Collects lexemes into **parse trees.**

  Intermediate code generator—Checks the program for semantic errors. Translates the program into an intermediate language, which often looks similar to assembly language.

  Optimizer—Attempts to improve programs by making them smaller or faster. May be omitted.

  Code generator—Translates intermediate code to machine language.

- All phases of a compiler use the symbol table, a data structure stores the names used in the program and their attributes.

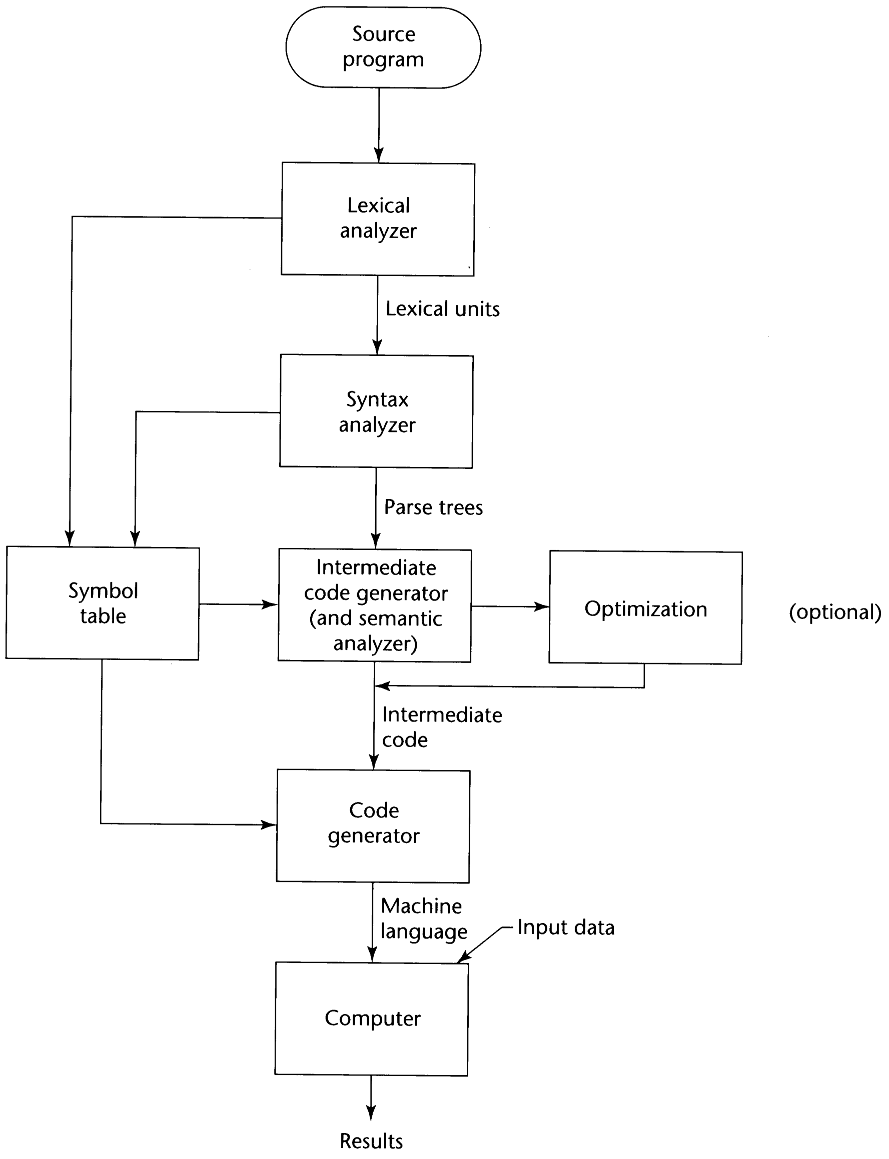
- Once a program has been compiled, many language implementation systems require the use of a **linker.** A linker has several responsibilities:

  Combine code from different modules that belong to a single program.
  Include code for calls of library routines.
  Include code for communication with the operating system.

  The output of the linker is a **load module** or **executable image.**

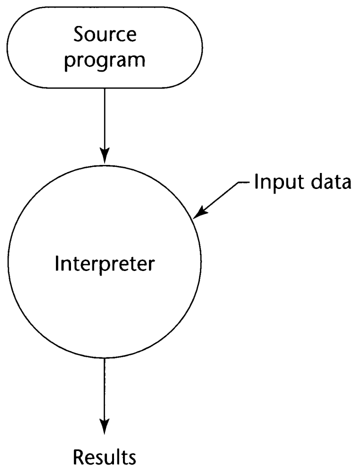
Compilation (Continued)

- Diagram of the compilation process:



Pure Interpretation

- Pure interpretation involves the direction execution of a program by another program, called an interpreter. With pure interpretation, the source program is not translated to a lower-level form.



- Advantages of interpretation:

  Good for debugging, since errors can be caught easily at run time and meaningful error messages displayed.

- Disadvantages of interpretation:

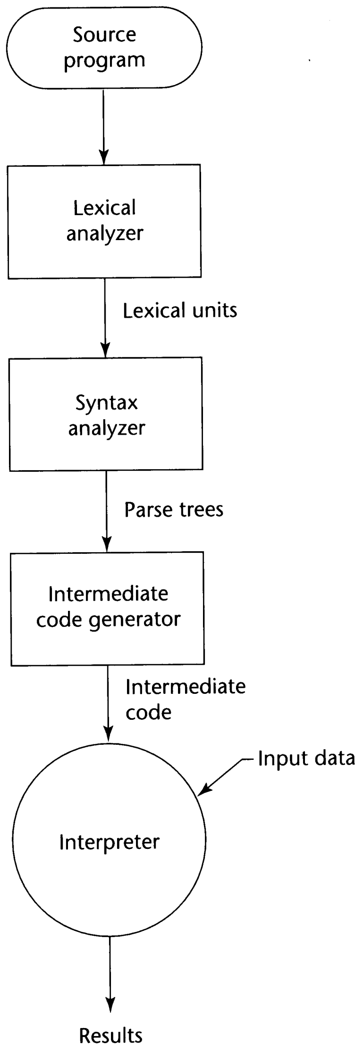
  Slow execution (10–100 times slower than a compiled program).
  Often requires more space during execution.

- Pure interpretation was used for some early languages (including APL, SNOBOL, and Lisp). By the 1980s, interpretation was relatively rare (although BASIC interpreters were common on PCs).

- Interpretation has recently made a comeback, thanks to Web scripting languages such as JavaScript and PHP.

Hybrid Implementation Systems

- A **hybrid implementation system** translates a source program into an intermediate language, which is then interpreted.



- Perl and Java both rely on a hybrid implementation system. A Java compiler produces **byte code** that is later interpreted by an implementation of the Java Virtual Machine.

- Java interpreters often perform Just-In-Time (JIT) compilation, translating byte code instructions into machine code instructions during program execution. Microsoft's .NET platform uses a similar approach.

Preprocessors

- C and C++ rely on a **preprocessor**, which adds an extra step to the compilation process.

- Preprocessor commands are embedded in the source code. The preprocessor executes these commands and removes them from the program prior to compilation.

- The following command causes the preprocessor to copy the file `myLib.h` into the source code of a program:

  ```
  #include "myLib.h"
  ```

- Preprocessor commands are also used to define macros:

  ```
  #define max(A, B) ((A) > (B) ? (A) : (B))
  ```

  The preprocessor will transform the statement

  ```
  x = max(2 * y, z / 1.73);
  ```

  into

  ```
  x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73));
  ```

Programming Environments

- A programming environment is a collection of tools used to develop software.

- A programming environment can be as simple as a text editor, a compiler, and a linker.

- An IDE (integrated development environment) is a more elaborate environment that provides a set of integrated tools accessed through a uniform user interface.

  Microsoft's Visual Studio supports several programming languages.