# 1. Introduction

## 1.1 The Art of Language Design

There are thousands of high-level programming languages, and new ones continue to emerge. Why?

1. **Evolution**: Find better ways to do something over time.
    - **GoTo-based** flow control: Basic, Fortran, Cobol, *evolved to*
    - **Repetition** (for, while, loops), **Selection** (if, else, switch, match), *evolved to*
    - **Nested block structure** (nested repetition and selection): Ada, Pascal, Algol, *evolved to*
    - **Object Orientated**: C++, Eiffel, Smalltalk
2. **Special Purpose**: Support a specific task or solve a specific problem.
    - **C** was developed to support creating a kernel and operating system development
    - **Java** was developed to support running the same program on "any device"
    - **Lisp** was developed for the implementation of A.I.
    - **Oracle** was developed for database.
    - **HTML** was developed for implementing a webpage.
3. **Personal Preference**: Every programming language has pros and cons, which are very subjective.
    - d

What makes a language successful?

1. **Expressive power**: Being able to express powerful concepts easily
    - c++: ++, --, +=, ...
    - visual basic: And <-> AndAlso, Or <-> OrElse
2. **Ease of Use**: People's view of how easy it is to program with.
3. **Ease of Implementation**: Unix C++ -> Visual C++

4. **Standardization**: If something is not standardized, then it causes massive headaches.
5. **Open Source**: Limiting access to something does not encourage people to use it.
    - ex. DoD for Ada, IBM Pl/I
6. **Excellent Compilers**
7. **Economic, Patronage, and Inertia**

It is a combinations of the above factors that may help determine whether a language is "good". We need to consider the *viewpoints* of both the *programmer* and the *language implementor*.

## 1.2 The Programming Language Spectrum

The many existing languages can be classified into families based on their **model of computation**.

- **Declarative**
    - *functional*: Lisp/Scheme, ML, Haskell
    - *dataflow*: Id, Val
    - *logic*, constraint-based: Prolog, spreadsheets, SQL

**Declarative** languages focus on *what the computer is to do*. More in tuned with *programmer's point of view*.

Example:

```
SELECT S.name
FROM Student S, Faculty F
WHERE S.major = 'CS' AND F.name = 'Charlie Shim'
```

- **Imperative**
    - *von Neumann*: C, Ada, Fortran, ...
    - *object-oriented*: Smalltalk, Eiffel, Java, ...
    - *scripting*: Perl, Python, PHP, ...

**Imperative** languages focus on *how the computer should do it*. Mainly for *performance reasons*.

## 1.3 Why Study Programming Languages?

1. A good understanding of languages design and implementation can help one **choose the most appropriate language for any given task**. Most languages are better for some things than others.
2. Make it **easier to learn new languages**. There are *basic concepts that underlie all programming languages*, such as
    - types
    - control (iteration, selection, recursion, nondeterminancy, concurrency)
    - abstraction
    - naming
3. Programmers with a strong grasp of language theory will be in a **better position to design elegant, well-structured notation** that meets the needs of current users and facilitates future development.

## 1.4 Compilation and Interpretation

### Question

Briefly explain the difference between compilation and interpretation. Give example. Are they mutually exclusive?

The compiler **translates** the high-level **source program into** an equivalent **target program** (typically in machine languages), and then goes away.

**Compilation generally leads to better performance**. In general, a decision made at compile time is a decision that does not need to be made at runtime.

Since the (final version of a) program is **compiled only once**, but generally **executed many times**, the savings can be substantial.

(Reference diagram 1)

| **Compilation** | **Interpretation** |
| --- | --- |
| Source compiled into target code via the Compiler, which is an executable. Then this executable is run with the input to create the output. | Interpretation uses a virtual machine to run the code directly with the input to produce output. |
| Specific to machine & O.S. | Independent of machine & O.S. |
| better performance | more flexible |
| more efficient (code optimization) | better debugging (generally) |