INTRODUCTION

References:

- Sethi, chapter 1
- Sebesta, chapters 1 and 2

Course Goals

Studying programming languages will help you to

- increase your vocabulary of programming constructs,
- read language manuals,
- learn new languages quickly,
- choose the right language for a task, and
- *design* a new language.

Course Themes

Principles of programming languages, including:

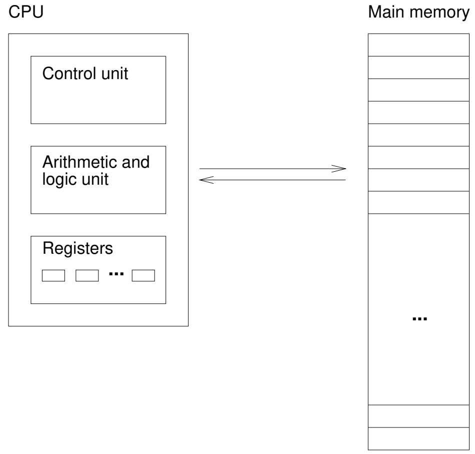
- scope and lifetime of variables
- parameter-passing techniques
- formalisms for describing the syntax of a language

Programming language **paradigms**, including:

- functional programming (exemplified by Lisp)
- logic programming (exemplified by Prolog)

Von Neumann Architecture

Most computers have the following basic structure:



(Named after John von Neumann, one of its originators.)

Memory is separate from the CPU, so instructions and data must be moved between memory and CPU.

The fetch-execute cycle

```
initialize the program counter
loop
   fetch the instruction pointed to by the ic
   increment the ic
   decode the instruction
   fetch needed data from memory, if any
   execute the instruction
   store the result
end loop
```

Executing an instruction is generally much slower than moving things between memory and CPU.

So the speed of this movement limits the speed of the computer.

This is the "von Neumann bottleneck".

Levels of programming language

Machine language

- Operations are very simple things like:

```
move contents of mem location 08125 to register 7
add contents of mem location 08125 to register 7
shift the contents of register 3 left one bit
jump to program line 85
skip the next instruction of register 1 is zero
```

- Instructions are encoded as numbers.
- No variables; operands are memory addresses or register numbers.
- Programming requires deep understanding of the machine architecture.
- Programs are not portable because instructions and their encoding are machine-specific.
- Programs are extremely hard to write, debug, and read.

Assembly language

- Operations and operands have symbolic names.
- Can use macros as shorthand for common sequences of code.
- An **assembler** translates into machine code.
- Still machine dependent.
- Almost as hard to write as machine code.

High-level language

- Examples: C, Lisp, Java, Fortran, ...
- Have higher level constructs. Example:

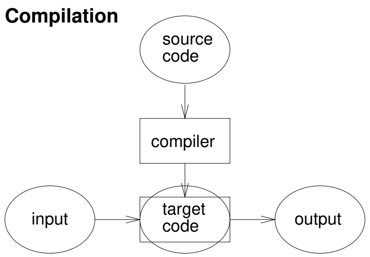
```
if (x == 3)              1 sub 3 5  ;sub 3 from reg 5
   <some instrns>        2 skp 5    ;skip if reg 5 is 0
else                     3 jmp 10   ;jump to line 10
   <other instrns>       4 <some instrns>
                        10 <other instrns>
```

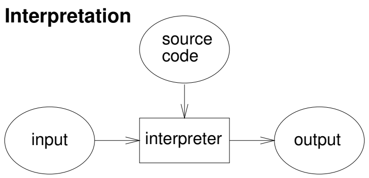
- Language usually supports type checking and other checks that help detect bugs.
- Programs are much easier to write, debug, and read.
- Programs are now machine independent.
- Programs may still be "implementation dependent".
- Before the first Fortran compiler (1957), it was commonly believed that any compiler would produce code so terribly inefficient as to be useless.

Translation

The process of converting a program written in a high-level language into machine language is called **Translation**.

There are two general methods.





Compilation: The whole program is translated before execution.

Interpretation: Translate and execute, one statement at a time.

Comparison of the two methods

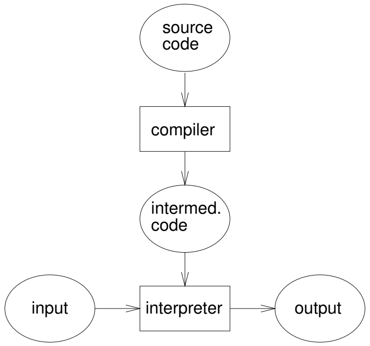
Compilation:

- Brings the program down to the level of the machine.
- Can execute translated program many times because the entire translation is produced.
- Program execution is much faster because the translator can do optimization.
- Harder to provide useful feedback when debugging because executing the target code.

Interpretation:

- Brings the machine up to the level of the program.
- Must re-translate for every execution.
- Program execution much is slower.
- Easier to provide useful feedback when debugging because executing the source code.

Pseudo-compilation: A hybrid of compilation and interpretation.



- A compiler translates the whole program before execution, but only into intermediate code.
- An interpreter translates and executes the intermediate code one statement at a time.
- The source program can be executed on any machine that has an interpreter for the intermediate code.

Java initially used this hybrid strategy. Its intermediate code is called **bytecode**.

Language Paradigms

Imperative languages

- Program statements are *commands*.
  Example: "Add 17 to x."
- Key operation: Assignment.
- Fits the von Neumann architecture closely.
- Examples: Fortran, C, Pascal, Turing.

Object-oriented languages

- Program describes communication between objects.
  Example: "Fraction f1, simplify yourself."
- Key operation: Message passing.
- Can also be imperative or functional.
- Examples: Simula, C++, OOTuring, Java.

Functional languages

- Program statements describe the value of expressions, using (essentially) *noun phrases*.
  Example:
  "The reverse of a list is last element followed by the reverse of the rest of the list (or is empty if the list is empty)."
- Key operation: Expression evaluation, by applying a function.
- Examples: Lisp, Scheme, ML

Logic-based languages

- Program statements describe facts and rules.
  Example:
  "Fact: Doug is Tom's father.
  Rule: If x is y's father and y is z's father, then x is z's grandfather."
- Programs don't say how to find a solution.
- Key operation: "Unification" (the how).
- Example: Prolog.

What Makes a Good Language

General goals:

- The language should be easy to learn.
- Programs written in it should be easy to write and to read.

Properties of a language that help meet these goals:

- Minimum number of concepts.
- "Orthogonality": concepts combine systematically, with no exceptions.
- Simple syntax.
- No synonyms.
- Meaning of a construct doesn't depend on context.

- Naturalness for the intended applications. Has the control structures, data structures, and operations, that are needed, and the syntax doesn't get in the way.

- Language not *too* concise (or programs will be too terse).

- Language concise enough (or programs will be too long).

- Has compile-time or run-time checking.

- Support for abstraction and information-hiding.

- Can be implemented efficiently.

- Portability.