Is Computer Programming a Science?

Hassan Aït-Kaci

MPR Teltech NSERC Chair
Intelligent Software Systems
Simon Fraser University
Canada

Lecture Outline

- Importance of programming
- A poser, for your enlightenment...
- What is computer programming?
- The Von Neumann model
- The prehistory of programming
- The quest for abstraction
- The C(++) phenomenon
- The struggle of the “how” and the “what”
- The mathematics of programming
- Programming computers of the future
- Conclusion: Is programming a Science?
- Bibliography

Importance of programming...

Fact: on July 22, 1962, a rocket carrying Mariner I -- an unmanned Venus probe -- had to be destroyed less than 5 minutes after being launched.

Loss: estimated at $20 million.

Cause: the ground-control computer program guiding the rocket should have contained:

```
IF NOT (in radar contact)
   THEN (do not correct flight path)
```

But, the **NOT** was inadvertently left out...

This very program had been previously used without problems for 4 lunar launches!

A poser?

For your enlightenment...

...and your entertainment

Given: a $m \times n$ array of whole numbers (i.e., a table of m rows, where each row is a sequence of n numbers).

Step 1: sort each of the m rows independently in ascending order.

Step 2: sort each of the n columns of the resulting array independently in ascending order.

Question: are the rows of the final array still sorted?

What is computer programming?

According to Bell Labs' Ravi Sethi [9]:

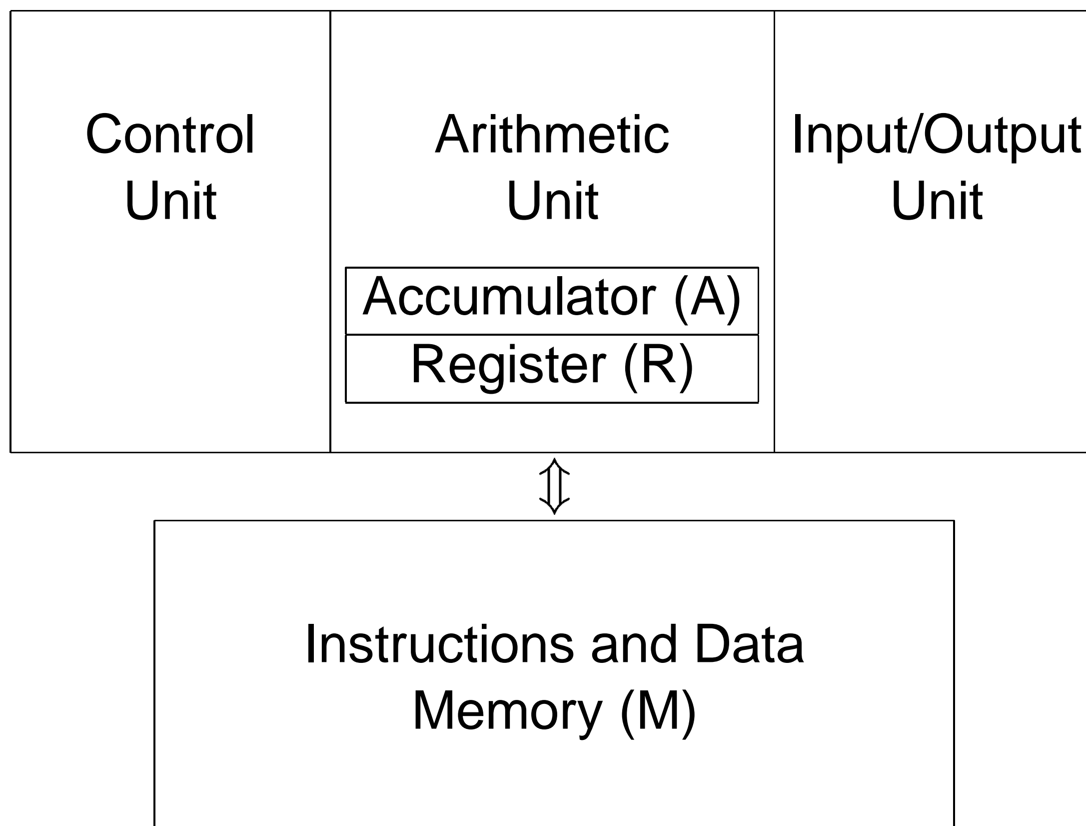
- A **program** is a specification of a computation.
- A **programming language** is a notation for writing programs.

The fundamental insight that makes mechanical computing possible is to have realized the duality principle whereby programs and data are the same!

Indeed: **(computable) functions are numbers!**

The Von Neumann Computer

The essential architecture of the modern computer is known as the "Von Neumann model"



Von Neumann Machine Instructions

- **Arithmetic:**

  $A \leftarrow A + M[i]$
  $A \leftarrow A - M[i]$
  $A \leftarrow A \times M[i]$
  $\ldots$
  $(A, R) \leftarrow (A \; \mathtt{mod} \; M[i], A \; \mathtt{div} \; M[i])$
  $\ldots$

- **Data Movement:**

  $A \leftarrow M[i]$
  $M[i] \leftarrow A$
  $R \leftarrow M[i]$
  $A \leftarrow R$

- **Control Flow:**

  $\mathtt{go\ to}\; M[i]$
  $\mathtt{if}\; A \geq 0 \;\mathtt{go\ to}\; M[i]$

The prehistory of programming

- Machine language
- Assembly language
- The FORTRAN insight

Machine language

Basically, a machine's basic cognitive unit is an on/off switch, mathematically encodable as 0/1.

Hence, all numbers may be encoded (in radix 2) and, by the duality principle, so can computable functions.

Machine Language: Any arbitrary encoding convention in radix 2 of the basic numbers and Von Neumann basic instructions.

E.g.,

```
000010111101010000001
001000000110000001100
000000100000011000001
```

could be used to encode the program:

load some number into accumulator
add some number to accumulator
store accumulator contents in register

Assembly language

Problem: Readability.

Solution: Mnemonic instructions.

E.g., write:	rather than:
`LOAD I`	`00001011110101000001`
`ADD J`	`00100000110000001100`
`STORE K`	`00000010000011000001`

A single program, called an (instruction) assembler, is needed to decode mnemonic instructions into binary code.

Advantage: readability.

Problem: portability.

The FORTRAN insight

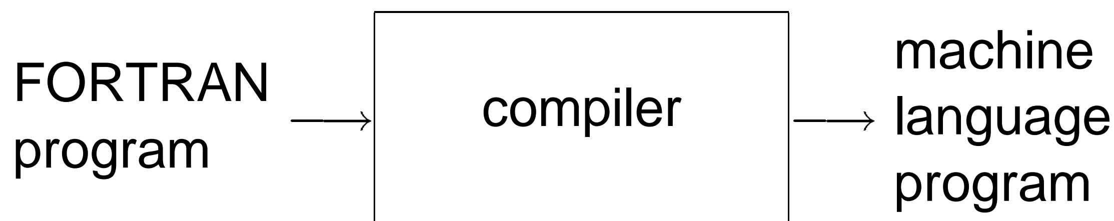
In the late 50's, IBM's John Backus suggested that writing a **formula** like:

$$K = I + J$$

was not only easier than:

```
LOAD I
ADD J
STORE K
```

but also could be executed on any machine provided a single (machine-dependent) program called a compiler was written for **FOR**mula **TRAN**slation:



The quest for abstraction

Clearly, the more abstract the notation, the easier the programming.

However, the more abstract the notation, the harder the translation.

Hence, higher abstraction seems to go against program efficiency!

Indeed, less fine-grained advantage may be taken of the specific machines.

But also, the lower the level, the more error-prone, and the harder to verify correctness.

The quest for abstraction (ctd.)

Question: Other abstractions since FORTRAN?

- **Control flow:**
  - imperative style (instruction-oriented, sequencing, iterative loops, . . .)
  - applicative style (expression-oriented, composition, recursion, . . .)
  - logical style (rule-oriented, conjunction, recursion, . . .)
  - concurrency (multi-tasking, process communication, distribution, . . .)
- **Data structuring:**
  - Structured types
  - Automatic memory management
  - Higher-order types
  - Abstract data types (encapsulation)
  - Polymorphism
  - Subtyping

The quest for abstraction (ctd.)

- **Program organization:**
  - -- Modularity
  - -- Genericity
  - -- Object-orientation
- **Program safety:**
  - -- Type checking
  - -- Abstract interpretation
  - -- Executable specification
  - -- Program verification

The C(++) tidal wave

A historical and sociological phenomenon.

- unplanned, but endemic proliferation
- simple, unpretentious, and very low-level
- vehicle of a most successful system (UNIX)

American permissive pragmatism...

vs.... **European formal austerity.**

- Everything's allowed - at your own risks!
- No type-checking, no memory management, no encapsulation
- Full operating system at finger tips
- Purists hate it; hackers love it!

The C(++) tidal wave (ctd.)

Program efficiency vs. Programming efficiency

- High-cost for prototyping
- Encourage “bad style”
- Unmaintainable

C++ = C with type-checking, encapsulation, and inheritance.

What vs. How: Holy Grail?

Procedural programming: tell *how* to proceed.

- efficient
- harder to verify and modify
- verbose and error-prone
- high investment to prototype

Declarative programming: state *what* is desired.

- harder to optimize
- easier to verify and modify
- more concise and clearer
- easy to prototype

What vs. How: Holy Grail? (ctd.)

- Functional Programming
  - + Simple & powerful mathematics (λ-Calculus)
  - + Formal and uniform: easier reasoning and transformability
  - + Higher-order
  - + Side-effect free, inherent parallelism
  - - Wasteful structures
  - - Unrealistic assumptions
- Logic Programming
  - + Simple & powerful (Predicate Calculus)
  - + Formal and uniform...
  - + Computation = non-deterministic logical inference
  - + Easy to embed constraint propagation
  - - High-overhead for deterministic programs
  - - Hard to understand for neophytes

The mathematics of programming

- Fundamental tools: discrete mathematics
  - -- Set theory
  - -- Formal logic
  - -- Universal algebra
  - -- Type theory
  - -- Category theory
- Formal semantics
  - -- Axiomatic
  - -- Denotational
  - -- Operational
- Formal correctness
  - -- *ex post facto:* program verification
  - -- *a priori:* specification refinement
  - -- *ex nihilo:* executable specification

The future of programming

Proliferating computers are bringing about the most profound revolution yet witnessed by humankind.

We, today's living generations, will see tremendous changes of every aspects of human life by the use of computers.

The key vehicle for controlling computers is *programming*. It is a formidable, though double-edged, weapon to wield.

Lest we become victims of an ever-growing anarchic maelstrom of *ad hoc* hackery, we must educate ourselves to systematic programming.

Is Programming a Science?

It should be!

...and it has, as all exact sciences do, a precise mathematics.

The mathematics of computer programming is wonderfully rich, yet barely explored and exploited.

The "vacuum call" of our present-day market exploitation of this magical technology encourages a **"program first, then (maybe too late) think"** culture, as opposed to a systematic **"think first, then design"** attitude that mature science and engineering requires.

Will we live up to the challenge?

...History will tell.

Bibliography

1. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
2. Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
3. Carl Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
4. Bruce MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Holt, Rinehart, and Winston, 1983.
5. Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
6. Benjamin Pierce. *Basic Category Theory for the Computer Scientists*. MIT Press, 1991.
7. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
8. David Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994.
9. Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.
10. Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.