Principles of Programming Language

[BE SE-6th Semester]

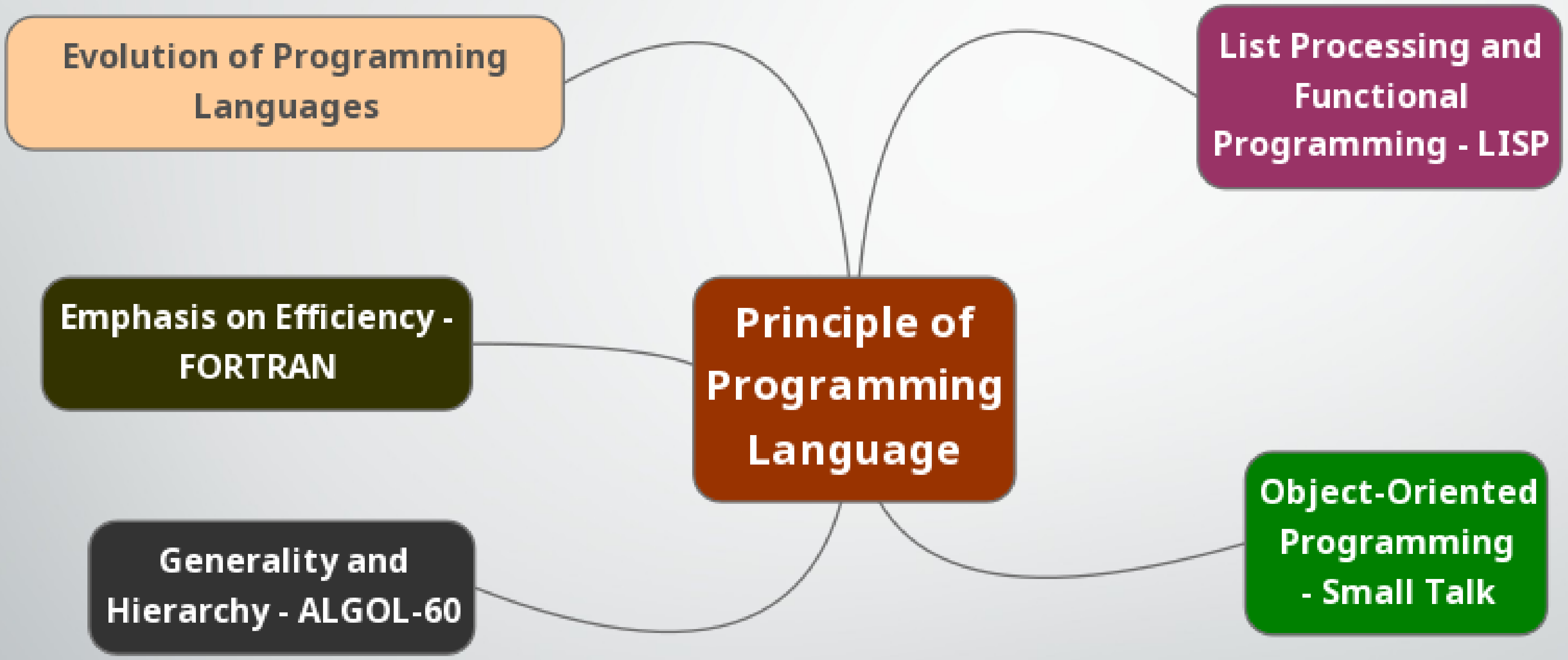
Rishi K. Marseni

Textbook:

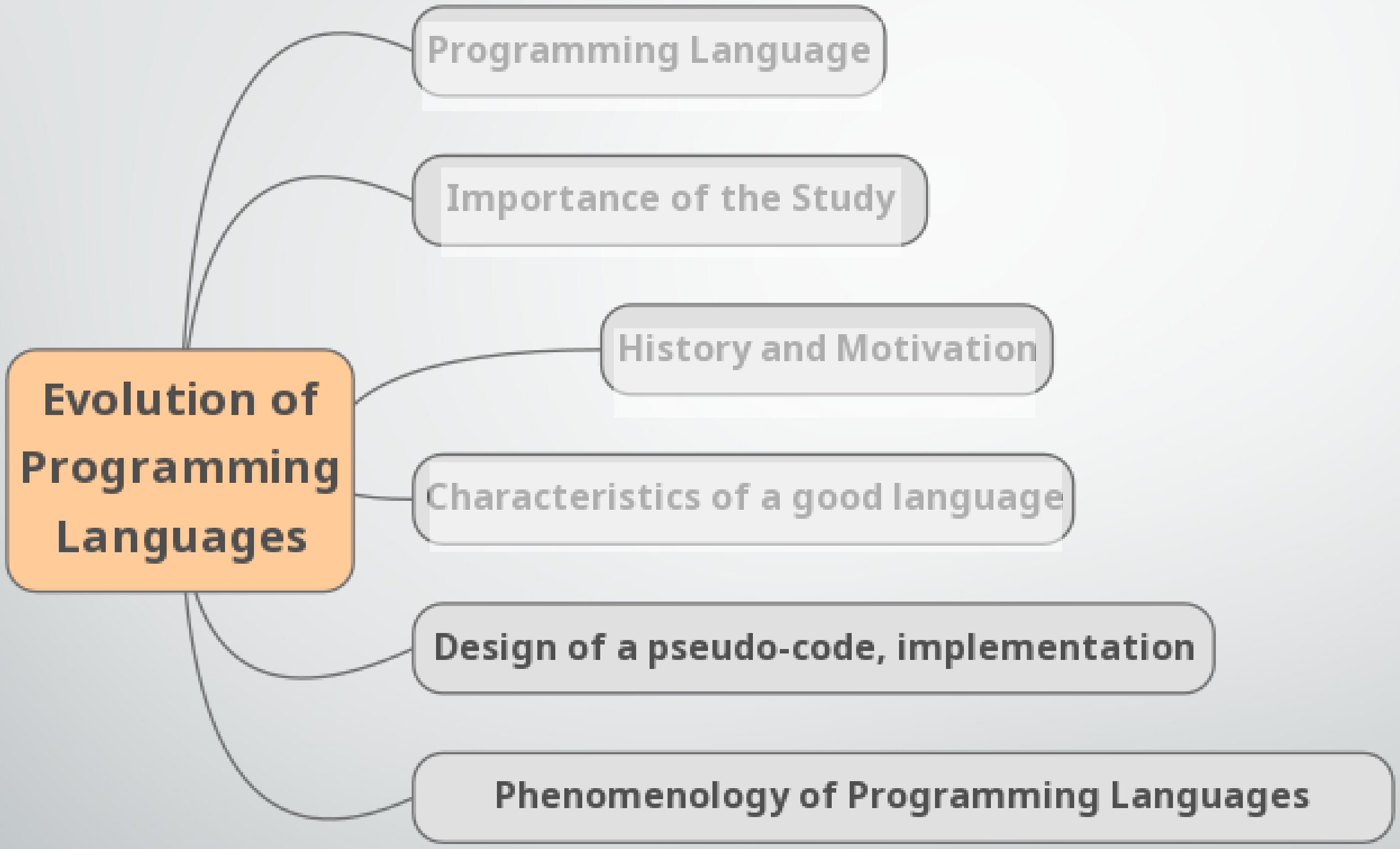
Principles of programming languages: design, evaluation, and implementation.

Author: Bruce J. MacLennan

Principle of Programming Language


Evolution of Programming Languages
Emphasis on Efficiency - FORTRAN
Generality and Hierarchy - ALGOL-60
Principle of Programming Language
List Processing and Functional Programming - LISP
Object-Oriented Programming - Small Talk

Unit 1: Evolution of Programming Language


Evolution of Programming Languages
Programming Language
Importance of the Study
History and Motivation
Characteristics of a good language
Design of a pseudo-code, implementation
Phenomenology of Programming Languages

Program DESIGN Notations

- Notations are used to resolve complexity
- Computer cannot understand notations directly
- Notations help programmer for designing:

  → memory layouts

  → control flow (flow diagrams/ flowcharts)

  → Mnemonics (instruction codes)
- High level(ease_of_use) vs Low level(efficiency)

Hardware limitations(1)

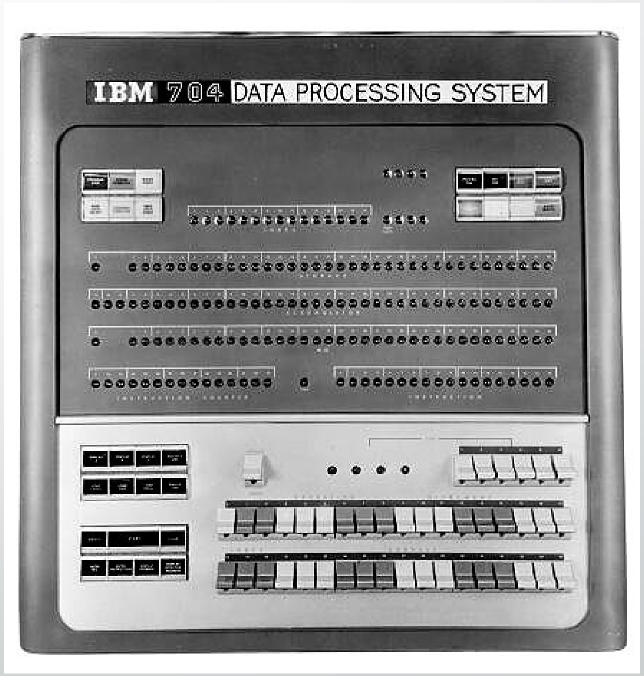
A. Floating Point Arithmetic

- Complex representation (harder to support)
- No direct representation was possible
- Manual scaling:

  → Multiply by constant factor

  → Use integer processor

  → Manually scale back result
- Complicated and error-prone process



Hardware limitations(2)

B. Indexing (Array as a popular data structures**)**

- Array = Adding a variable index quantity to a fixed address
- Indexing was not supported by early computers
- Before array(address modification techniques):
  → Altering the program’s own data accessing instruction
  → Compute actual address from pointer and offset, then write into instruction’s data address portion
  → separate address modification code is needed
- Very error prone process > we need interpreters

Pseudo-Code

- An instruction code that is different than that provided by the machine
- Pseudo-code offered floating point support and indexing
- Has an interpretive subroutine to execute
- Implements a virtual computer:
  - Has own data types and operations(instruction sets)
  - Higher level than actual hardware
  - Provides facilities more suitable to applications
  - Abstracts away hardware details

Need of Pseudo code

- During first generation of computer, programming was very difficult
- A programmer need to know about the hardware specification of every machine
- For example in 1950’s for IBM 650 which has following characteristics:
    - No programming language was available (not even assembler)
    - Memory was only a few thousand words.
    - Stored program and data on rotating drum.
    - Instructions included address of next instruction so that rotating drum was under next instruction to execute and no full rotations were wasted.

The subroutine

- The subroutine is an important part of any architecture
- A subroutine is a group of instructions that usually performs one task
- It is a reusable section of the software that is stored in memory once, but used as often as necessary
- e.g. multiply, on machines which did not have a 'multiply' instruction
- One copy of the subroutine could thus be 'shared' among multiple uses.

Pseudo-code Interpreters

- a primitive, interpreted programming language
- an interpretive subroutine developed to run the pseudo-code
- Implements a virtual computer(add-on data types & operations)
- simulated instructions not provided by the hardware
- an example of “Automation Principle” of programming language
- commonly used to perform floating-point operations & indexing

Design of a Pseudo-Code

Based on the capabilities and constraints of the first generation computers.

In 1950, Capabilities expected by the programmers and not support by the hardware at that time are:

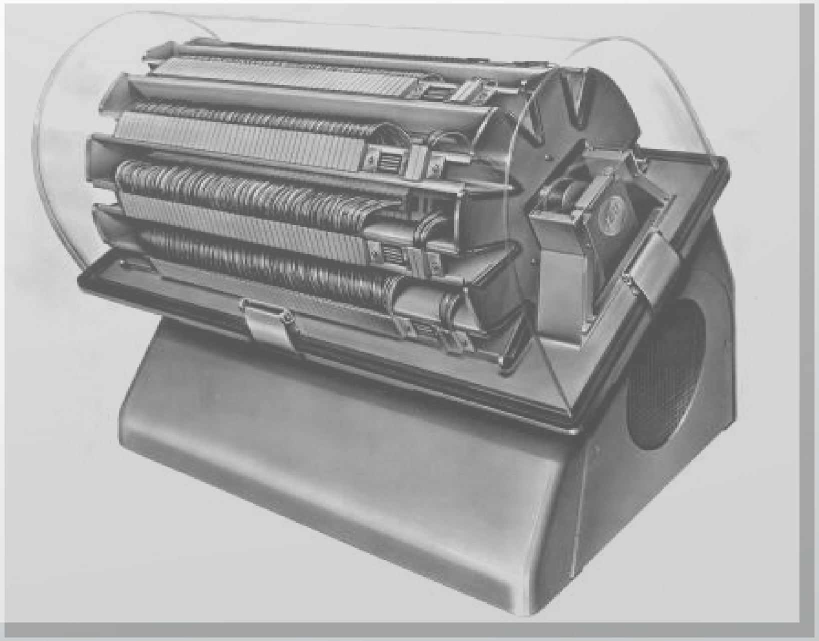
- Floating point operation support (+,-,*,/,…)
- Comparisons (=,≠,<,≤,>,≥)
- Indexing
- Transfer of control
- Input/output

Hardware Assumptions

The IBM 650 will serve as the hardware

- 1 word: 10 decimal digits + 1 sign
- 2000 bytes memory

- 1000 bytes for data
- 1000 bytes for program



7-Principles of Pseudo-code(1)

The Automation Principle

- Automate mechanical, tedious, or error prone activities.

The Regularity Principle

- Regular rules, without exceptions, are easier to learn, use, describe, and implement

Impossible error principle

- Making errors impossible to commit is preferable to detecting them after their commission

Orthogonality principle

- Independent functions should be controlled by independent mechanisms

7-Principles of Pseudo-code(2)

The abstraction Principle

- Avoid requiring something to be stated more than once; factor out the recurring pattern

Labeling principle

- Do not require users to know absolute numbers or addresses. Instead associate labels with number or addresses

Security principle

- No program that violates the definition of the language, or its own intended structure, should escape detection.

Language Design

1 word can be enough to specify a 3-operand instruction

- Operation: sign + 1 digit => Supports 20 operations
- Operands: 3 3-digit operands => Each accessing memory locations in data area
- Use the sign to get more orthogonality

Orthogonal design:

Operations should be more intuitive than machine code

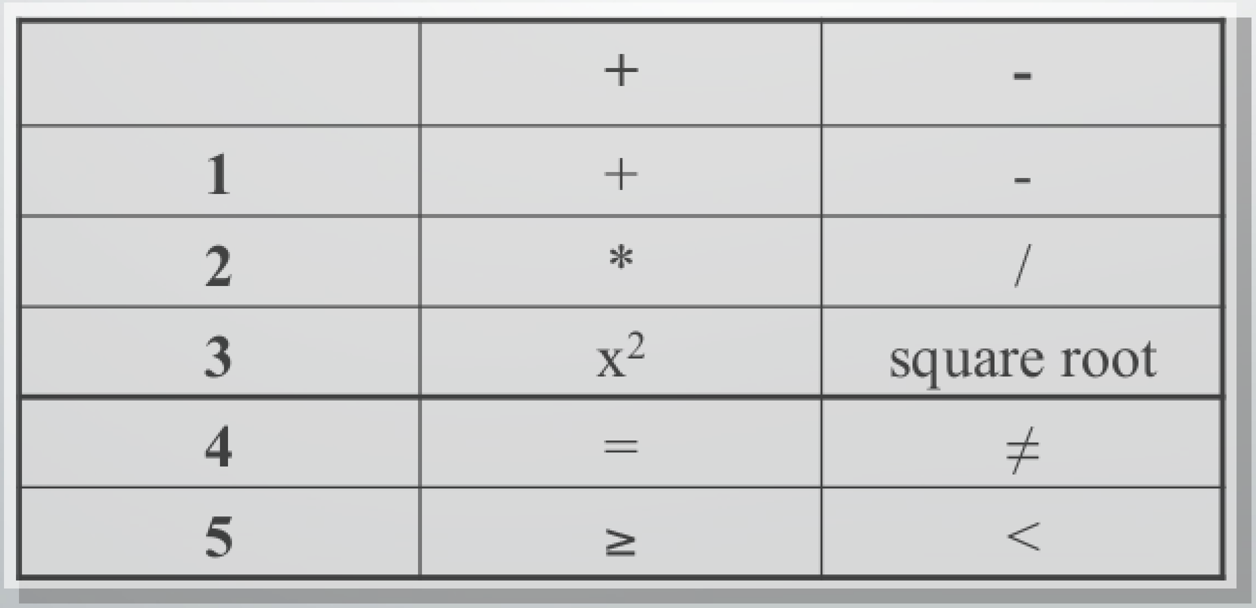
Pseudo-code Instruction format(1)

1. Arithmetic Operations

Operand src1 src2 dst

E.g.: x + y → z : +1 010 150 200

“Add values at location 010 and 150, and save it to location 200”



	+	-
1	+	-
2	*	/
3	x^2	square root
4	=	≠
5	≥	<

Pseudo-code Instruction format(2)

2. Comparisons

Operand src1 src2 dst

if x < y then go to z

First 2 operands are data locations, dst is address of next instruction

3. Moving

Operand src1 000 dst

+ 0 150 000 200

First operand is source, third operand is destination while second operand is not used

Pseudo-code Instruction format(3)

4. Indexing

Pseudo-code was provided with built-in indexing

Operand base_address index dst

+6 xxx iii zzz [Get: $x_i \rightarrow z$]
-6 xxx yyy iii [Put: $x \rightarrow y_i$]

- **Examples:**
  if there is a 100-element array beginning at location 250 in data memory, and location 050 contains 17

+6 250 050 803 : move the contents of location 267(=250+17) to location 803

-6 722 250 050 : move the contents of location 722 to location 267

Pseudo-code Instruction format(4)

5. Looping

Looping through the elements of an array is frequently used

Looping instruction

– Iterator variable (array index i)

– Upper bound (n)

– Address of beginning of loop (d)

– “+7 iii nnn ddd”

The operation increments location iii and loops to instruction ddd if the result is less than the contents of nnn.

Pseudo-code Instruction format(5)

6. Input / Output

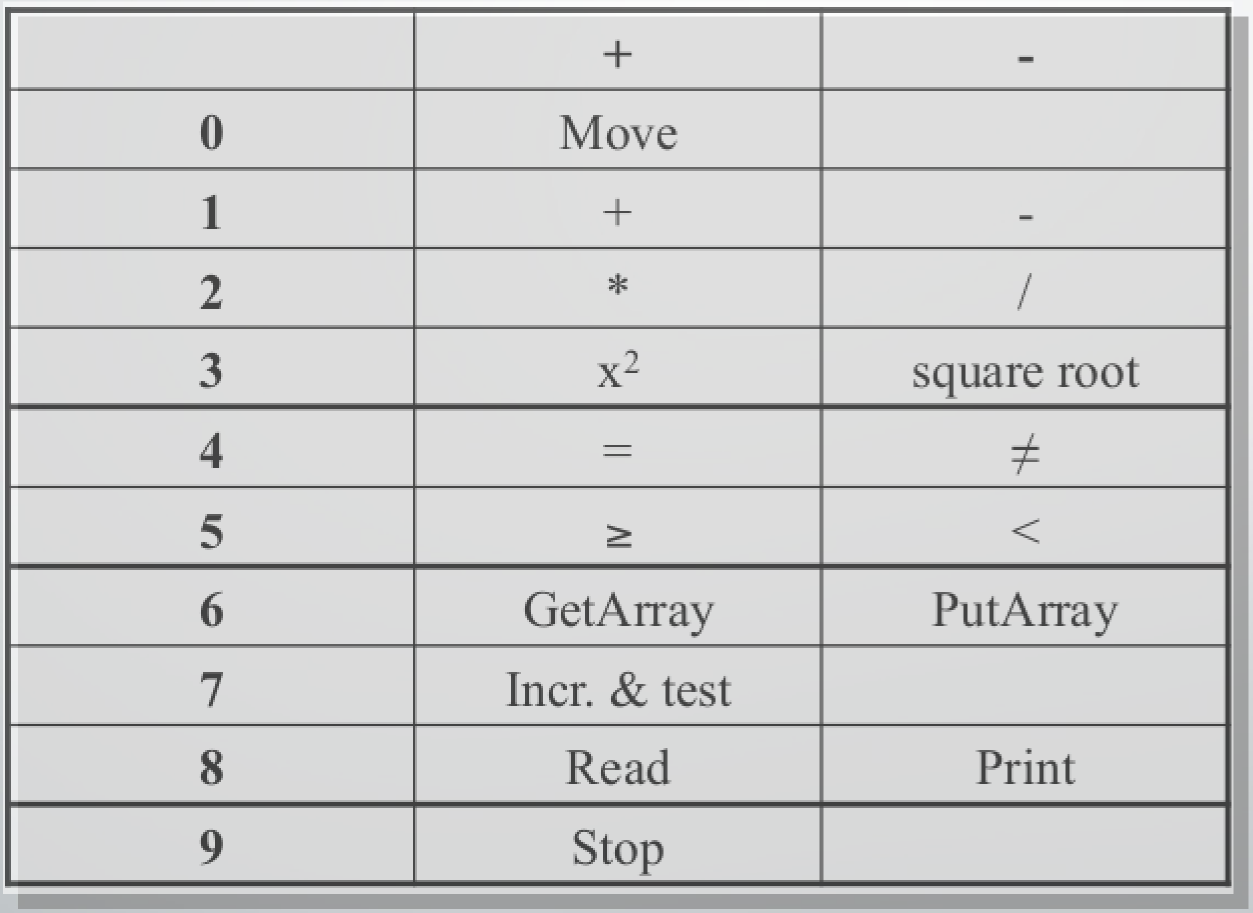
- Program needs to read data from input and write data to output.
- Needs only a memory location to read from or write to

Read: “+8 000 000 dst”

Print: “-8 000 000 src”

Pseudo-code Instruction format(6)

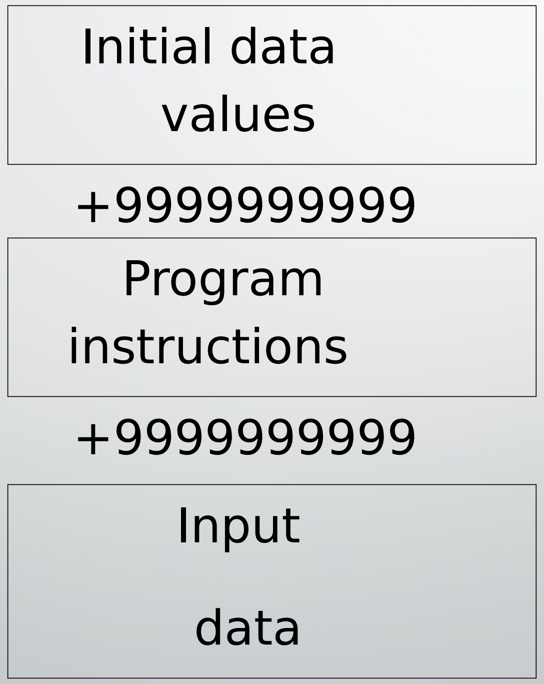
Complete Pseudo-code operations



	+	-
0	Move	
1	+	-
2	*	/
3	x^2	square root
4	=	≠
5	≥	<
6	GetArray	PutArray
7	Incr. & test	
8	Read	Print
9	Stop	

Pseudo-code Program Structure

- a means of constructing the program as a whole
- interpreter read initialization cards and their content in consecutive memory locations



Implementing the Interpreter

An interpreter for pseudo-coded program can be implemented as:

- Model interpreter behavior after manual execution
- Cheat: Implement using a high-level language
- We have to simulate the hardware in software

The second option is impossible as high level language is not developed during 1950s.

We will focus on the third implementation method.

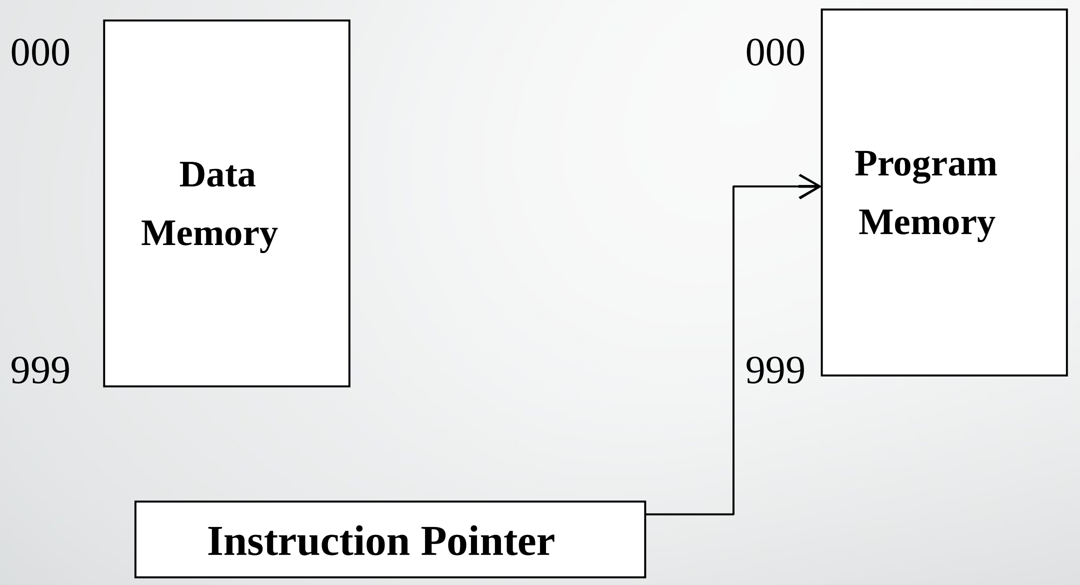
Data Structures

Data structures are needed to simulate the IBM 650

IBM 650 contains the following data structures:

- Data memory
- Program memory
- Instruction pointer

Structure of the Interpreter



The Read-Execute Cycle

- The instruction pointer is updated after each read operation
- Increment in step 1; overwrite if needed

instruction : = Program[IP];   IP : = IP+1;

- The Read-Execute Cycle:

1) Read the next instruction
2) Decode the instruction
3) Execute the operation
4) Continue from step 1

Decoding Instructions

- designed with a regular structure, decoding is simple
- extract the sign, operation code, and the three address fields

dest := abs (instruction) mod 1000

The operations types are:

1) Select operation → Switch-statement (case-statement)
2) Arithmetic operations → Straight-forward
3) Control-flow → IP may also need to be altered

Pseudo-code Labeling

- make programs readable (non executable codes)
- labels are used for replacing absolute addressing in the programs
- label definition operator:

- 7 0LL 000 000

- symbolic notation **LABL nn**
- the control flow instructions allow jump to labels instead of absolute address

+ 4 xxx yyy 0LL

Interpreting Labels(1)

- Look through all instructions from beginning of program?

  → Yes, but that is slow | like interpreters

- Create label table with absolute addresses for labels and bind addresses

  → Much faster | Compilers do it this way

Label	**Location**
20	001
40	005
50	009

Interpreting Labels(2)

Labeling guideline:

Check all the labels are defined once

if there is any referencing to undefined labels

if a label is defined in more than one place

Use the Label tables

Data Labels

- Variables can be processed like labels
- We can have variable declarations in Initial-data section of the program

0 sss nnn 000

- Declare a storage area with symbolic name sss, nnn locations long, initialized to all
- Symbolic notation **VAR sss nnn**
- If nnn=1 then simple variable else(nnn>1) then array

Data Declaration

- **Example 1:** **0 111 001 000**
  0 000 000 000

  A simple variable, labeled 111, is declared and initialized to zero

- **Example 2:** **0 666 150 000**
  3 141 592 654

  Declares a 150-element array, identified by the label 666, Initialized to all 3141592654

Data Bindings

- For each declaration, the loader keeps track of the next available memory location, and binds the symbolic variable number to that location.
- Binding time of the declaration is load time

X:=X+10

Value | Type | Valid range of values | Set of possible types
Representation of the constant 10 | Properties of operator

- Binding time :
  { Execution | Translation | Language implementation | Language definition}

Debugging

- Debugging always has to be done
- It can be facilitated for debugging by printing instructions executed in order
- Interpreter can include trace, to get a trace of the execution of the program
- Trace is a record of the instructions it has executed

Complete Symbolic Language

	+	-
0	move MOVE	
1	+ ADD	- SUB
2	* MULT	/ DIV
3	X2 SQR	square root SQRT
4	= EQ	≠ NE
5	≥ GE	< LT
6	GetArray GETA	PutArray PUTA
7	Incr. & test LOOP	Label LABL
8	input READ	output PRNT
9	end STOP	Trace TRAC

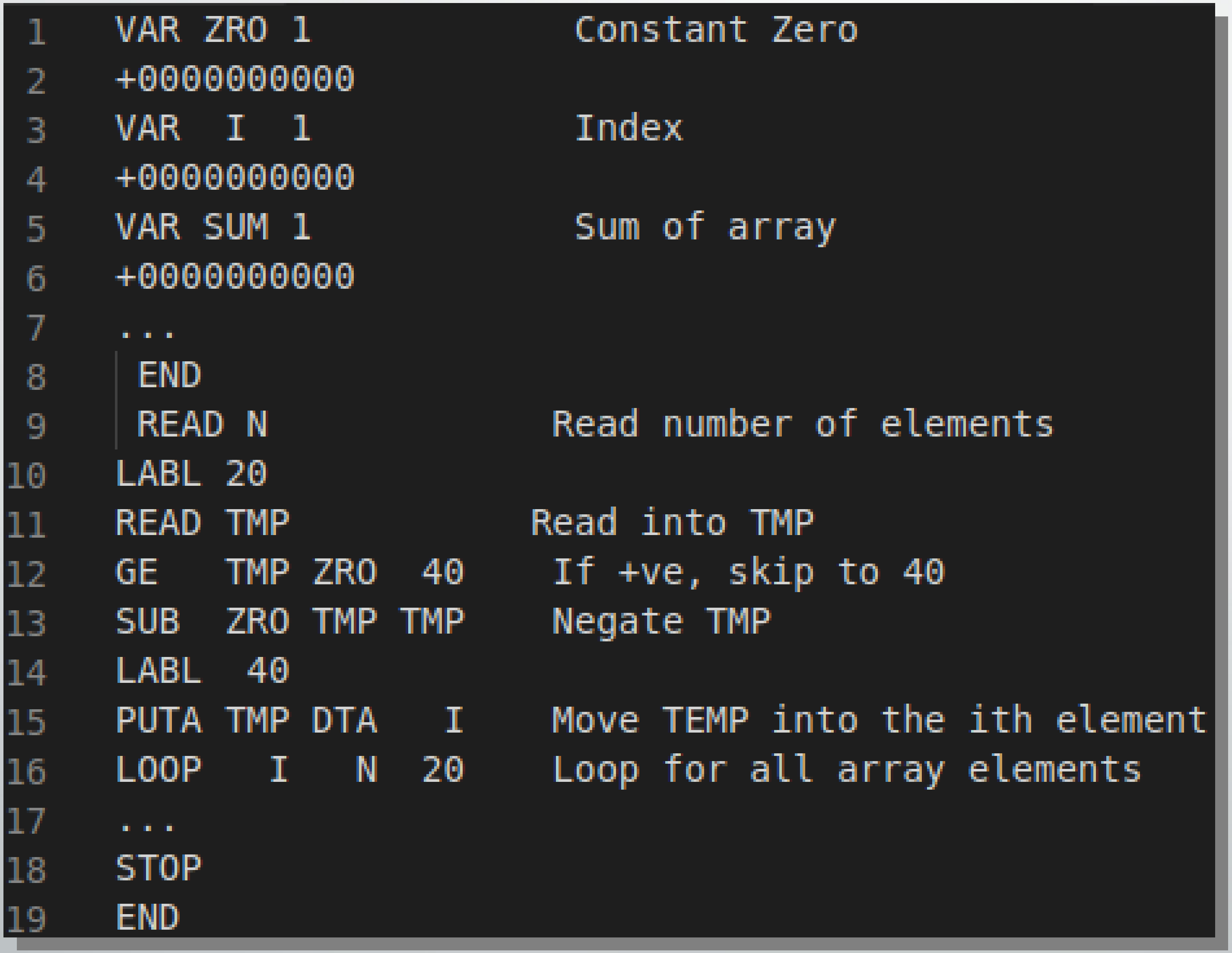
Additional symbols

- LABL nn
  - Declare label *n*
- VAR sss nnn
  - Declare variable *s*[*n*]
- END
  - Delimiter between variables, program and input
  - Defined as -9999999999
- TRAC
  - Enable/disable tracing
  - Tracing is turned off by default. Encountering this operation toggles tracing.

Implement a symbolic pseudo-code

- The interpreter can record in the symbol table, the size of the array and so,can check each reference to the array
- Prevents a violation of the program’s intended structure {Security Principle}
- symbol table for operations and operands
- A program array for encoded instruction
- The loader performs a translation function

Sample Program



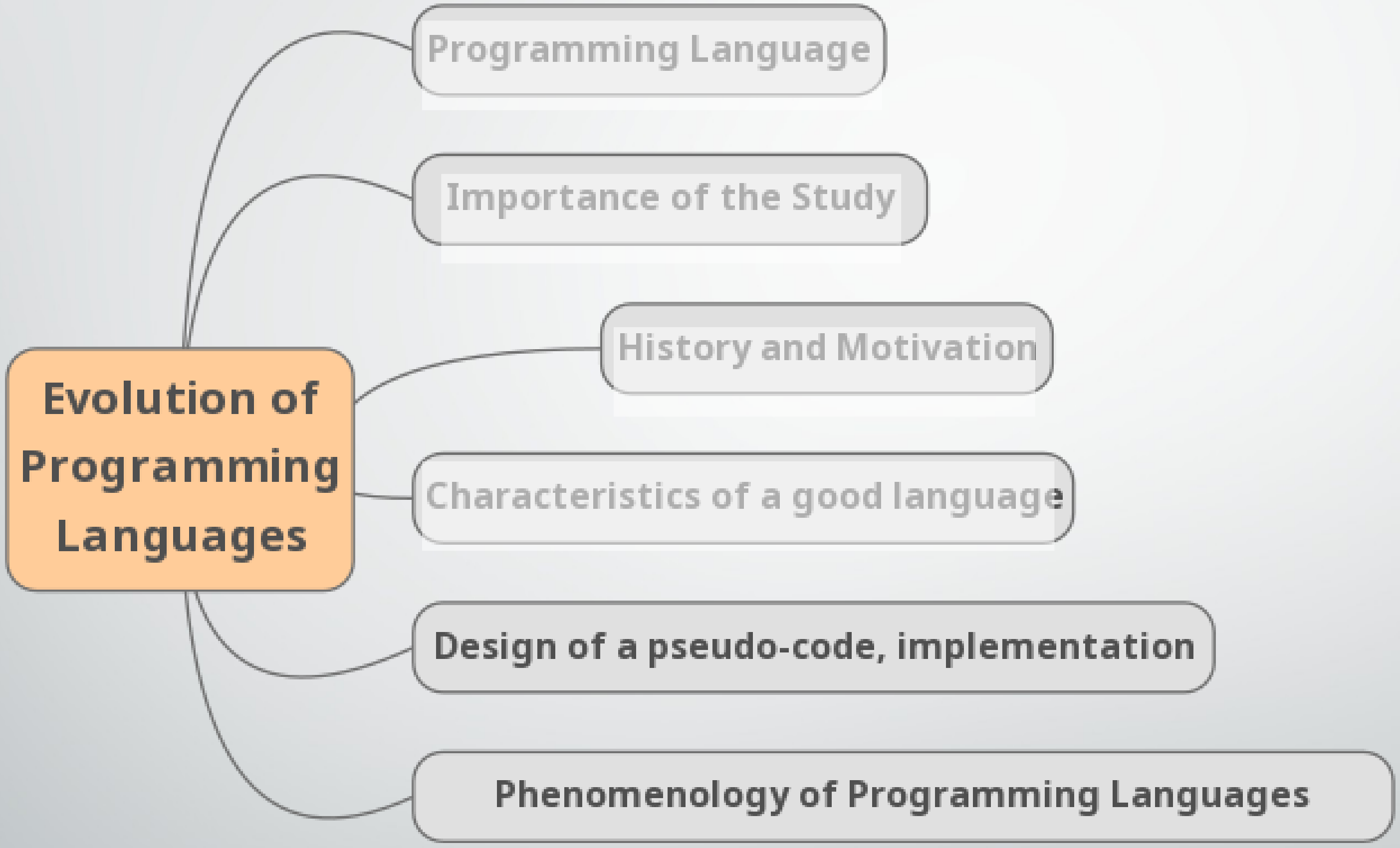
```
VAR ZRO 1                Constant Zero
+0000000000
VAR  I  1                Index
+0000000000
VAR SUM 1                Sum of array
+0000000000
...
 END
 READ N                 Read number of elements
LABL 20
READ TMP               Read into TMP
GE   TMP ZRO  40        If +ve, skip to 40
SUB  ZRO TMP TMP        Negate TMP
LABL  40
PUTA TMP DTA   I        Move TEMP into the ith element
LOOP   I   N  20        Loop for all array elements
...
STOP
END
```

Programming Languages as Tools
[Phenomenology **]**

- Tools are both amplifying and reductive
- A stick to knock the fruit down, instead of your hands
- Fascination and fear are common reactions to new tools
- With mastery, you no more feel the tool as an external, additional object
- Programming languages influence, focus and action

  Writing by pen, a typewriter, a word processor

Unit 1: Evolution of Programming Language


Evolution of Programming Languages
Programming Language
Importance of the Study
History and Motivation
Characteristics of a good language
Design of a pseudo-code, implementation
Phenomenology of Programming Languages