Pseudo-Code

CS4100

Dr. Martin

From *Principles of Programming Languages: Design, Evaluation, and Implementation (Third Edition)*, by Bruce J. MacLennan, Chapter 1, and based on slides by Istvan Jonyer.

1

Role of programming languages

- What is a *programming language*?
  - Formal Method
    - Describe a solution to a problem
    - Organize a solution to a problem
    - Reason about a solution to a problem
  - Interface between user and machine
    - Trade-off
      - Ease of use - high level
      - Efficiency - low level

2

Role of programming languages

- What is a *programming language*?
  - A language that is intended for the expression of computer programs and that is capable of expressing any computer program.

3

Readability

- Is machine code readable?
  - 00000010101111001101010101011110
- Assembly language?
  - mov dx tmp
  - add ax bx dx
- Is high-level code readable?
  - http://www0.us.ioccc.org/years.html#2004
    - http://www0.us.ioccc.org/2004/arachnid.c
    - http://www0.us.ioccc.org/2004/anonymous.c

4

Pseudo-Code

- An instruction code that is different than that provided by the machine
- Has an interpretive subroutine to execute
- Implements a virtual computer
  - Has own data types and operations
- (Can view all programming languages this way)

5

Pseudo-Code Interpreters

- Is programming difficult?
- In the 1950's, it was…
  - E.g.: IBM 650
    - 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
    - Problem: What if address is already occupied?

6

Part of an IBM 650 program

```
LOC  OP DATA INST  COMMENTS
1107 46 1112 1061  Shall the loop box be used?
1061 30 0003 1019
1019 20 1023 1026  Store C.
1026 60 8003 1033
1033 30 0003 1041
1041 20 1045 1048  Store B.
1048 60 8003 1105
1105 30 0003 1063
1063 44 1067 1076  Is an 02-operation called for?
1076 10 1020 8003
8003 69 8002 1061  Go to an 01-subroutine.
```

7

Program DESIGN Notations

- Complexity led to development of *program design notations*
  - Memory layout
  - Control flow
    - *Flow Diagrams* (von Neumann & Goldstine)
    - Later: Flowcharts
  - Mnemonics
    - To help remember instruction codes
    - Like assembly language today
- These were designed to help the programmer, not to be interpreted by computers

8

Floating Point Arithmetic

- Earliest built-in floating point processing: IBM 704
- Before that, it had to be *simulated*
  - Manual scaling
    - Multiply by constant factor
    - Use integer processor
    - Manually scale back result
    - Complicated and error-prone process

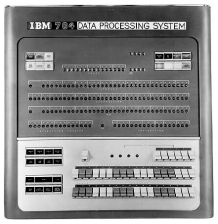
9

- IBM 650 and card reader

http://www-03.ibm.com/ibm/history/exhibits/650/650_album.html

IBM 704 Operator's Console

http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_2423PH704C.html



10

Indexing

- Array is one of most common data structures
- Indexing
  - "Adding a variable index quantity to a fixed address in order to access the element of an array"
  - Indexing was not supported by early computers
  - They used address modification
    - Alter the program's own data accessing instruction
    - Compute actual address from pointer and offset, then write into instruction's data address portion
  - Very error prone process

11

Pseudo-Code Interpreters

- Subroutines were commonly used to perform floating-point operations and indexing
- Consistent use of these simplified the programming process
- This simulated instructions not provided by the hardware
- Next logical step:
  - Use instruction set not provided by the computer
  - Pseudo-Code interpreter (a primitive, interpreted programming language)

12

"Appendix D"

- Why not simplify programming by providing an entire new instruction code that was simpler to use than the machine's own.
- Wilkes, Wheeler and Gill (1951) describe a pseudo-code and an "interpretive subroutine" for executing it
  - Buried in the now famous Appendix D of *The Preparation of Programs for an Electronic Digital Computer*
  - They must have not realized the significance of their work…

13

A Virtual Computer

- Pseudo-code interpreters implement
  - A virtual computer
  - New instruction set
  - New data structures
- Virtual computer:
  - Higher level than actual hardware
    - Provides facilities more suitable to applications
    - Abstracts away hardware details

14

Principles of Programming

- 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.

15

Design of a Pseudo-Code

- Remember: it's 1950!
- Capabilities we want
  - Floating point operation support (+,-,*,/,…)
  - Comparisons (=,≠,<,≤,>,≥)
  - Indexing
  - Transfer of control
  - Input/output

16

Hardware Assumptions

- The IBM 650 will serve as the hardware
  - 1 word: 10 decimal digits + 1 sign
  - 2000 byte memory
    - 1000 for data
    - 1000 for program

http://www-03.ibm.com/ibm/history/exhibits/650/650_intro2.html

Principles of Programming

- Impossible error principle
  - Making errors impossible to commit is preferable to detecting them after their commission.
  - E.g.: Cannot modify the program accidentally, since memory modifying operations are for "data memory" only

18

Language Design

- 1 word can be enough to specify a 3-operand instruction
  - Operation: sign + 1 digit
    - Supports 20 operations
  - 3 3-digit operands
    - Each accessing memory locations in data area
  - Orthogonal design:
    - Operations should be more intuitive than machine code
    - Use the *sign* to get more orthogonality

19

Principles of Programming

- Orthogonality principle
  - Independent functions should be controlled by independent mechanisms.

20

Specifics

- Instruction format:
  - op 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"
  - Orthogonal design: subtract should be '-1'

21

Arithmetic Operations

	+	-
1	+	-
2	*	/
3	x^2	square root

22

Comparisons

- Comparisons alter control flow
  - `if x < y then go to z`
  - First 2 operands are data locations, *dst* is address of next instruction

23

Extended Instruction Table

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

24

What else do we need?

- Moving
  - Could do "add 0" to an address, but that could be inefficient
  - Dedicate an operation to moving
  - Second operand is not used
  - "+0 src 000 dst"

25

Indexing

- Need
  - Base address
  - Index
- Base and index take up 2 operands; what can we do with 3rd?
  - Save value of indexed element for other operations
- Index operations:
  - Get: $x_i \rightarrow z$: `+6 xxx iii zzz`
  - Put: $x \rightarrow y_i$: `-6 xxx yyy iii`

26

Looping

- Looping through the elements of an array is frequently used
- What's needed?
  - Iterator variable (array index *i*)
  - Upper bound (*n*)
  - Address of beginning of loop (*d*)
  - "`+7 iii nnn ddd`"

27

Principles of Programming

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

28

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`"

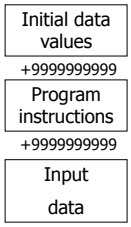
29

Complete Instruction Set

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

30

Program Structure



31

Implementing the Interpreter

- How to implement the interpreter for our pseudo-coded program?
  - Model interpreter behavior after manual execution
  - Cheat: Implement using a high-level language ☺
  - We have to simulate the hardware in software

32

Data Structures

- What data structures are needed to simulate the IBM 650?
  - Data memory
  - Program memory
  - Instruction pointer

33

Structure of the Interpreter

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

- Where do we update the instruction pointer (IP)?
  - Step 4? No: we may need to jump, which would be overwritten
  - Increment in step 1; overwrite if needed

34

Decoding Instructions

- Extract part of instruction
  - `dst = instruction mod 1000`
- Select operation
  - Big switch-statement (case-statement)
- Arithmetic operations
  - Straight-forward
- Control-flow
  - IP may also need to be altered

35

Labeling

- What if we need to insert an instruction?
  - All addresses would have to be shifted, and the code updated
- Solution:
  - Use labels for loops, instead of absolute memory addresses
  - Define label:
    - `-7 0LL 000 000`
    - Only 100 numeric labels are possible (00-99)
  - Modify control flow instructions to jump to labels

36

Interpreting Labels

- How do we handle labels in the interpreter?
  - Look through all instructions from beginning of program?
    - Yes, but that is slow. This is how some interpreters work. (BASIC, for instance)
  - Create label table with absolute addresses for labels and bind addresses
    - Much faster. Compilers do it this way.

37

Principles of Programming

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

38

Data Labels?

- If we can jump to a label, we could use labels for variables as well
- Construct symbol table
- This idea is easily extended to instructions as well to form a symbolic pseudo-code

39

Data Declaration

- We could extend the language to include symbols not only for program instructions but for data declarations as well
- In initial data values:

```
+0 sss nnn 000
±ddddddddddd
```

  - Declare *n* values of *d* referenced by symbol *s*
  - Symbolic notation:

```
VAR sss nnn
±ddddddddddd
```

  - n=1 : simple variable
  - n>1 : array

40

Debugging?

- Debugging always has to be done…
- Can facilitate debugging by printing instructions executed in order
- Interpreter can include *trace* flag

```
if trace is enabled
  print IP, instruction
```

41

Complete Symbolic Language

	+	-
0	move MOVE	
1	+ ADD	- SUB
2	* MULT	/ DIV
3	X^2 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

42

Complete Symbolic Language

- 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.

43

Sample Program

```
VAR ZRO 1
+0000000000
VAR I 1
+0000000000
VAR SUM 1
+0000000000
...
END
READ N
LABL 20
READ TMP
GE   TMP ZRO  40
SUB  ZRO TMP TMP
LABL  40
PUTA TMP DTA   I
LOOP   I   N  20
...
STOP
END
+0000000005
+0000000020
...
```

44

Principles of Programming

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

45