Introduction to programming

Video lecture - Introduction to programming and software

Relevance

- Introduce you to programming
- Give you an idea about
  - Instructions - what the CPU can do and how to instruct it to do so
  - Memory - how to store and retrieve values from named memories
- Show you the TOY computer instruction set and programming
- Show you what various programming languages look like
- Introduce the term Software
- Introduce the term Algorithm and show you an algorithm for searching

Requirements

We expect that you have read the wiki, read Kernighan (D is for digital): Inside the CPU... Part II - Software, pages 35-116, and that you have read the Swedish compendium in relevant parts.

CPU

- The CPU is the central processing unit of your computer
- Does all of the work (most of it anyway)
- Can’t do so many things, but do them well and fast
- Controls hardware communication
- Arithmetics
- Reading and writing to the memory
- Has an instruction set (the things it can do)
- ***Can compare numbers and decide what to do next***

The TOY computer is a model of a computer

- It has a processor with an instruction set
- It has a memory
- It can be programmed using instructions
- It has an accumulator (special memory)

The TOY computer is an example from Brian Kernighan:
http://www.kernighan.org/toysim.html

TOY instruction set

```
get         read a value from user to accumulator
store M     store the current value to location M
load Val    update the accumulator with value Val or value
            at location Val
ifzero L    if the accumulator is zero, jump to label L
add Val     update the accumulator by adding Val to it or
            adding the value at location Val to it
sub Val     update the accumulator by subtracting Val from it
            or subtracting the value at location Val from it
goto L      jump to label L
print       print the contents of the accumulator
stop        end execution
```

Accumulator

- The CPU uses a special memory for storing intermediate values

```
Accumulator: [ ]
Accumulator: [10]
Accumulator: [5]
Accumulator: [5]
Accumulator: [10]
Accumulator: [10]
Accumulator: [0]
Accumulator: [0]
```

```
   instruction          memory M
← load 10
← sub 5
← store M        →        5
← add M          →        5
← store M        →       10
← sub M          →       10
← ifzero Label   →       10
  (program jumps to Label)
```

Small program

```
Main get        ← Main is a label, get reads from user to acc.
 ifzero End     ← If accumulator is 0, jump
 add sum        ← Load value @ sum to accumulator
 store sum      ← Write accumulator's value to sum
 goto Main      ← Jump to label Main
End load sum    ← End is a label, load sum to acc.
 print          ← Print accumulator's value to screen
 stop           ← End execution
sum 0           ← Memory "sum" is declared at the end
```

So, does the computer speak English?

- No. It “speaks” numbers.
- All instructions are numbered, e.g.:

GET=1, PRINT=2, STORE=3, LOAD=4, ADD=5, STOP=6, GOTO=7, IFZERO=8

- The program could now be written as a sequence of numbers
- That’s why variables (named memories like sum) come at the end
- The computer could now load the numbers and interpret them as instructions

Numeric version of the program

	Main								End				Sum
	GET	IFZERO	END	ADD	Sum	STORE	Sum	GOTO	Main	LOAD	Sum	PRINT	STOP
Instr:	1	8	10	5	14	3	14	7	1	4	14	2	6	0
Mem addr.:	1	2	3	4	5	6	7	8	9	10	11	12	13	14

When the program executes, Sum will change as will the accumulator. The rest is just jumping around in memory until we reach STOP.

Same program in Bash

```
sum=0
read num
while ((num != 0))
do
  sum=$((sum + num))
  read num
done
echo "Sum is: $sum";

# Higher level of abstraction!
```

Same program in Java

- Install Java if you want to try:
  `sudo apt-get update && sudo apt-get install openjdk-9-jdk`

```
public class Adder {
  public static void main(String[] args) {
    int num;
    int sum = 0;
    while ( (num = Integer.parseInt(System.console().readLine())) != 0 ) {
      sum += num;
    }
    System.out.println("Sum: " + sum);
  }
}
```

Java needs compilation and the JVM

```
$ javac Adder.java && java Adder
10 ← user enters 10
20 ← user enters 20
0  ← user enters 0
Sum: 30 ← Java prints
```

Same program in Python

```
import sys
num = int(input())
sum = 0
sum += num

while num != 0:
  num = int(input())
  sum += num

print ("Sum: ", sum )
```

Same program in Python

```
$ python3 adder.py
10 ← user enters 10
20 ← user enters 20
0  ← user enters 0
Sum: 30 ← Python prints
$
```

Software

- Programmers write software
- A program is written in some programming language in a plain text file
- Some programming languages must be translated to machine code (numbers that the computer understands) using a compiler - *Compiled languages*
- Other languages can be run by an interpreter for the language - Interpreted languages
- And then, there's Java (and some similar languages) which is first compiled to bytecode, then interpreted - The java command interprets bytecode

Compiled languages

[source code file] → compiler → [executable binary for the OS and computer]

- C, C++
- Pascal
- Ada
- It’s a two-step process (simplified). First compile, then run.
- The executable works only on your computer and OS combo
- You can cross-compile - compile on your computer for my computer
- The (compiled) program is binary

Interpreted languages

[source code file] is executed by an interpreter

- Python
- Perl
- Bash
- BASIC
- Lisp
- JavaScript
- Same source code can be run by interpreters on many platforms
- The program is plain text

Bytecode compiled languages

[sourc code file] → compiler → [bytecode file] → interpreted by runtime

- Java, Scala, Groovy, Clojure
- .NET (a whole bunch of languages)
- The compiled bytecode will run on any platform that has a runtime/interpreter
- The bytecode is binary

Algorithms

- An algorithm is a careful, precise and unambiguous description of how to solve a (also carefully, precise and unambiguous description of a) problem
- An algorithm can be *implemented* in a programming language, so that computers can use the algorithm to solve the problem
- Should be guaranteed to terminate (with the problem solved)
- Typical algorithms include
  - Sorting
  - Searching
  - Finding the shortest way

Binary search algorithm

- Fast way to search a sorted list for an element
- Utilizes the fact that the list is sorted
- Starts at the middle, compares the wanted element to the found
  - If the wanted element is larger than the found, then discard all larger elements and start over
  - Otherwise discard the other half, and start over
- Makes the problem half the size each lookup

Example

List: [12, 20, 33, 42, 55, 70, 88, 102, 110]

You want to see if 20 is in the list:

- Start at middle - 55
- 20 is less than 55, so keep only the first half: [12, 20, 33, 42] and start over
- pick either 20 or 33 as middle (let’s be unlucky and pick 33)
- 20 is less than 33, so keep first half again [12, 20]
- Next “middle” happens to be 20, so we found it

Example

List: [1, 2, ..., 1000]

You want to see if 127 is in the list:

- Start at middle - 501, 127 is less than 501, so keep only the first half: [1, 2, ..., 500] and start over
- next “middle” is 251, so keep first half again [1,2,.. 250]
- Next “middle” is 126, so keep last half [127,128,...250]
- Next “middle” is 189, so keep first half [127, 128, ...,188]
- Next “middle” is 158, so keep first half [127,...,157]
- Next “middle” is 142, so keep first half [127,...,141]
- Next “middle” is 134, so keep first half [127,...,133]
- Next “middle” is 130, so keep first half [127,...,129]
- Next “middle” is 127, so keep first half [127]
- Only one element and it matched! Found it in 10 steps

How many steps does N elements take?

- Boils down to, how many times can you take half of N?
- Which is the same as what is the smallest power of 2, that is equal to or greater than N?
- Or, what power of 2 is exactly N?
- Or Log_2N
- Example: We have 1 099 511 627 776 elements.
- Now, 2^{40} happens to be exactly that number. So 40 steps.
- We have 1 000 000 000 000 elements. 2^{40} is the smallest power of two that is greater than or equal to 1 000 000 000 000.
- $Log_2$1 000 000 000 is 39.9. Round up to 40.

Binary search doesn’t only work with numbers

- Anything that can be sorted works
- A sorted list can be made of any class of elements that have a natural order (the concept of less than, equal or greater than)
- Text and words are common examples (use the alphabet/ASCII table)

The end