Introduction to programming - Summary

Software, programming and the CPU

CPU - Central processing unit

- Can do arithmetics (addition, subtraction, division and multiplication)
- Fetch values from the memory, store values
- Control hardware
- Make decisions on what to do next

The TOY computer (by Kernigan)

- http://www.kernighan.org/toysim.html Try it!
- Simulates a computer - so that you can program it
- Limited instruction set (things “it can do”)
- Demonstrates the basic operations of a CPU

Read and add numbers until user enters 0

```
Main get
 ifzero End
 add sum
 store sum
 goto Main
End load sum
 print
 stop
sum 0
```

Accumulator

- When we do arithmetics, we need somewhere to store intermediate results
- A register is a memory location inside the CPU (faster than main memory)
- Operations in the TOY involving the accumulator:
  - get - read value from user, store in accumulator
  - print - print the value of the accumulator to screen
  - load *Val* - load *Val* into the accumulator (without changing *Val*) *(a number or memory)*
  - store M - store contents of accumulator to memory M (accumulator keeps its value)
  - add *Val* - add *Val* to the current content of the accumulator (without changing *Val*)
  - sub *Val* - subtract *Val* from the current content of the accumulator (without changing *Val*)
  - ifpos L - go to instruction L if current content of the accumulator >= 0
  - ifzero L - go to instruction L if current content of the accumulator = 0

Memory

- For storing values longer time - referred to by variable names (memory locations)
- Not as fast as registers (but faster than e.g. hard drive)
- Operations in the TOY computer involving memory:
  - M Num - Before the program starts, store numeric value Num in memory location M
    - You can call it something else than M
    - Declared at the end of the program
  - load *Val* - load *Val* into the accumulator (without changing *Val*) *(a number **or memory**)*
  - store M - store contents of accumulator to memory M (accumulator keeps its value)
  - add *Val* - add *Val* to the current content of the accumulator (without changing *Val*)
  - sub *Val* - subtract *Val* from the current content of the accumulator (without changing *Val*)

Read and add numbers until user enters 0

```
Main get          ← Label Main, get number to acc
 ifzero End       ← If acc is 0, jump to End
 add sum          ← Add value from memory sum to acc
 store sum        ← Write acc value to sum
 goto Main        ← Jump to Main
End load sum      ← Label End, load sum to acc
 print            ← Print value of acc to screen
 stop             ← Stop execution
sum 0             ← Before execution begints, store 0
                    in memory sum
```

Same program in Bash, using a while loop

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

Same program in Bash, using a while loop

```
sum=0                       # ← memory/variable called sum
read num                    # ← read value from user to num
while ((num != 0))          # ← as long as num isn't 0...
do                          # ← do the following:
  sum=$((sum + num))        # ← store sum + num in sum
  read num                  # ← read new value into num
done                        # ← jump to while test
echo "Sum is: $sum";        # ← When while test fails, we do this

# read and echo are commands
# while, do, and, done are keywords
```

Same program in Java, using a while loop

```
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);
  }
}
```

Same program in Java, v.2., using a while loop

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

Same program in Java, v.3., using a while loop

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

Same program in Python using a while loop

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

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

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

Software, programming languages

- Programs are written in a programming language (by and for humans)
- Programs cannot be executed directly by the CPU
- Two main types of languages to solve this:
  - Compiled languages
  - Interpreted languages
- And, then there’s Java - both compiled and interpreted

Compiled languages

- C, C++, Ada and Pascal (and many more)
- Translates the source code to executable code for your OS and platform
- The translation is called “compilation” - so you need a compiler (and often a few other tools) to make your program executable
- Source code is written in plain text files
- When you compile, you compile for a platform (by default the one you are on)
- Compiled code is not portable (needs to be compiled for target platform)

Interpreted languages

- Bash, Python, and Perl (and many more)
- Source code is written in plain text files
- Source code can be interpreted by a special program, which then executes the instructions line-by-line
- You need the interpreter for your native platform, in order to interpret (execute) the program on your platform
- The source code is portable (as long as there’s an interpreter for the target platform)

Java - Compiled code is “portable”

- Source code is written in plain text files (with unicode support)
- Source code is compiled to an intermediate format - bytecode
- The bytecode is interpreted by a special program, the Java Virtual Machine (JVM)
- Two step process:
  - Compile to source code: `javac MyProgram.java`
  - Invoke JVM: `java MyProgram`
- As long as the target platform has a native JVM, the compiled code (bytecode) is portable

Algorithms

- Specific and unambiguous description of how to solve a specific and unambiguous problem
- Do not run the world - people running programs that encode algorithms might, but don’t blame the algorithm
- Must be guaranteed to terminate (otherwise it doesn’t “solve” the problem)
- Typical examples involve searching and sorting

Binary search

- Find an element in a sorted list (e.g. the phone book, a dictionary etc)
- Very efficient
- Works by making the problem half until it finds the element (if it’s there)
- Start at the middle
  - Is the wanted element before or after?
  - Look only in the half that should have the element - discard the rest
- Every time, start in the “new” middle and discard the half which can’t have the element

How efficient is binary search?

- Every lookup removes half the remaining list
- How many times can you divide a number by two?
- Hint: recall logarithms from school
- Let’s take 1 000 000 elements as an example
- How many times can you divide 1 000 000 by 2, before you reach 0?

.....

How efficient is binary search?

- 2^{20} is 1 048 576 - let's say this is the size of the list ~ one million
- So, you need at most 20 lookups in a sorted list of one million elements
  - First division discards 524 288 elements,
  - next discards 262144 elements,
  - next discards 131072 elements,
  - next discards 65536 elements,
  - next discards 32768 elements,
  - next discards 16384 elements,
  - next discards 8192 elements,
  - next discards 4096 elements,
  - next discards 2048 elements,
  - next discards 1024 elements,
  - etc (you can divide 1024 only ten times)